PPE User's Manual

Table of Contents

1. Preface

This document contains implementation-specific information about the PowerPC Processor Element (PPE). The PowerPC Processor Element (PPE) consists of the PowerPC Processor Unit (PPU) and the PowerPC Processor Storage Subsystem (PPSS).

1.1. Reference Materials

This document is intended for use in conjunction with the other supporting documents listed below.

- CBE Overview
- SPE User's Manual
- Cell Broadband Engine™ Architecture
- PowerPC Architecture Book, Version 2.02
    - Book I: PowerPC User Instruction Set Architecture, Version 2.02
    - Book II: PowerPC Virtual Environment Architecture, Version 2.02
    - Book III: PowerPC Operating Environment Architecture, Version 2.02

  (International Business Machines Corporation, 01/28/2005)
  http://www.ibm.com/developerworks/eserver/library/es-archguide-v2.html
- PowerPC Microprocessor Family: Vector/SIMD Multimedia Extension Technology Programming Environments Manual Version 2.06c

  (International Business Machines Corporation, 09/30/2005)
  http://www.ibm.com/chips/techlib/techlib.nsf/techdocs/C40E4C6133B31EE8872570B500791108
- Cell Broadband Engine™ Registers

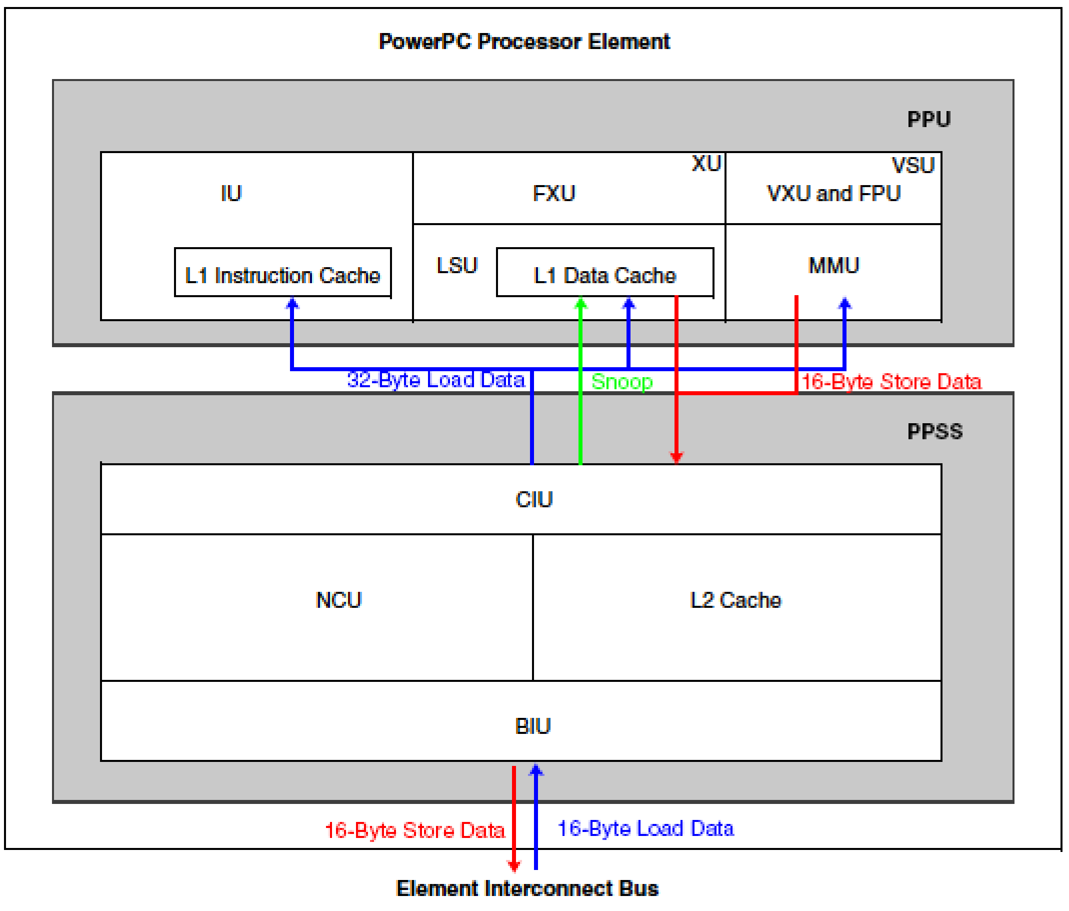
(Note) In the URLs listed in this document for reference, it has been verified that the documents can be referred to as of August 14, 2006. Please note that the pages may be moved or the contents of the pages may be changed after that.

2. PowerPC Processor Element (PPE) Overview

The PowerPC Processor Element (PPE) is a modern, general-purpose, high-frequency, 64-bit processor that delivers state of the art performance comparable to high-end, personal-computing processors. It can be combined into a larger system configuration of processors to form a top-of-the-line gaming system, a graphic workstation, a network processor, and so forth.

The PPE is designed with a paradigm of modularity. The intent is that separate subunits of the PPE can be reused in future designs without necessarily reusing the entire PPE. Thus, the PPE is comprised of a hierarchy of units and subunits (see *Figure 2-1*). At the top level, the PPE is split into two units called the PowerPC Processor Unit (PPU) and the PowerPC Processor Storage Subsystem (PPSS).

Figure 2-1. PPE High-Level Block Diagram



IU	Instruction unit	FPU	Floating-point integer unit
L1	Level 1	MMU	Memory management unit
XU	Execution unit	CIU	Core interface unit
FXU	Fixed-point integer unit	NCU	Noncacheable unit
LSU	Load and store unit	L2	Level 2
VSU	Vector/scalar unit	BIU	Bus interface unit
VXU	Vector media extension unit	PPU	PowerPC processor unit
		PPSS	PowerPC processor storage subsystem

2.1. PowerPC Processor Unit (PPU)

The PowerPC Processor Unit (PPU) deals with instruction control and execution. The instructions defined in *Book I: PowerPC User Instruction Set Architecture* and *Book III: PowerPC Operating Environment Architecture* are implemented in the PPU. It consists of the following five subunits (which themselves are further subdivided into logic blocks):

- Instruction Unit (IU)
  This unit is responsible for instruction control, including fetch, decode, dispatch, issue, branch, and completion. It also includes the level 1 (L1) instruction cache.
- Fixed-Point Integer Unit (FXU)
  This unit is responsible for integer code execution, including execution of add, multiply, divide, compare, shift, rotate, and logical instructions.
- Load and Store Unit (LSU)
  This unit is responsible for all data accesses, including execution of load and store instructions. It also includes the level 1 (L1) data cache.
- Vector/Scalar Unit (VSU)
  This unit is subdivided into a Floating-Point Unit (FPU) and a Vector/SIMD Multimedia Extension Unit (VXU). The FPU implements the instructions defined in *Chapter 4, Floating-Point Processor of Book I: PowerPC User Instruction Set Architecture, Version 2.02*. The VXU implements the Vector/SIMD Multimedia Extension instructions that are defined in *PowerPC Microprocessor Family: Vector/SIMD Multimedia Extension Technology Programming Environments Manual* Version 2.06c.
- Memory Management Unit (MMU)
  This unit manages both instruction and data address translation. It contains the Segment Lookaside Buffer (SLB) and Translation Lookaside Buffer (TLB).

2.2. PowerPC Processor Storage Subsystem (PPSS)

The PowerPC Processor Storage Subsystem (PPSS) handles memory requests (both data and instructions) that come out of the PPU. It also handles Element Interconnect Bus (EIB) requests that come into the PPE from other processors or I/O devices. The PPSS implements the instructions defined in *Book II: PowerPC Virtual Environment Architecture*. It consists of four subunits:

- Core Interface Unit (CIU)
  This unit serves as a throttling point and crossbar for communication between the PPU and the PPSS. It is essentially a collection of queues. It also handles data prefetch.
- Noncacheable Unit (NCU)
  This unit handles all noncacheable memory requests. It is essentially a collection of queues.
- Level 2 Cache Unit (L2) (shared for Instruction and Data)
  This unit implements the control and dataflow of the L2 Cache.
- Bus Interface Unit (BIU)
  This unit handles bus arbitration and pacing onto the EIB. It is essentially a collection of queues.

With the exception of the throttling queues in the CIU, all of the PPSS logic runs at half the frequency of the PPU logic.

2.3. Interface Between the PPU and the PPSS

A matrix of busses connects the various subunits in the PPE. These busses allow the subunits to communicate with each other as necessary. All traffic between the PPSS and PPU is sent through the CIU, whose primary job is to direct and pace all communication between these units. Connecting the PPSS and PPU are the following major busses:

- A 32-byte load data port (shared for MMU, L1 Instruction Cache, and L1 Data Cache reload requests)
- A 16-byte store data port (shared for MMU and L1 Data Cache store requests)
- A snoop bus from the L2 Cache to the L1 Data Cache

2.4. Interface Between the PPE and the Element Interconnect Bus

All communication between the PPE and the rest of the chip flows through the BIU. In addition to a number of control signals, the following major busses are implemented:

- A 16-byte load data port
- A 16-byte store data port

3. Data Format

3.1. Integer Data Format & Byte Ordering

The PowerPC User Instruction Set Architecture supports both big-endian and little-endian byte-ordering modes. A complete description of these modes is given in *Book I: PowerPC User Instruction Set Architecture*. For more information about big-endian and little-endian byte ordering, see the following section in that book:

- 5.3 Little-Endian

The *Cell Broadband Engine™ Architecture* supports only big-endian byte ordering. Therefore, PowerPC Processor Elements (PPEs) in a CBEA-compliant implementation are not required to support the little-endian byte-ordering mode defined in the PowerPC architecture. Synergistic Processor units (SPUs) do not implement the optional little-endian byte-ordering mode. The PPE structure mapping is identical to that used for the SPUs in a CBEA-compliant system. Since the *Cell Broadband Engine™ Architecture* supports only big-endian byte ordering, the memory flow controller (MFC) DMA command and control registers do not implement the optional little-endian byte-ordering mode.

The DMA data transfers themselves are simply byte moves, without regard to the numerical significance of any byte. Thus, the big-endian or little-endian issue becomes irrelevant to the actual movement of a block of data. The byte-order mapping only becomes significant when data is fetched or interpreted (for example by a processor or by an MFC).

Structure Mapping Example provides an example of PPE structure mapping that uses big-endian byte ordering. The mapping shown is identical for the SPUs and the MFCs in a CBEA-compliant system.

Structure Mapping Example

Figure 3-1 shows an example of a C language structure 's' which contains an assortment of scalars and one character string. The value assumed to be in each structure element is shown in hexadecimal in the C comments. These values show how the bytes that make up each structure element are mapped into storage.

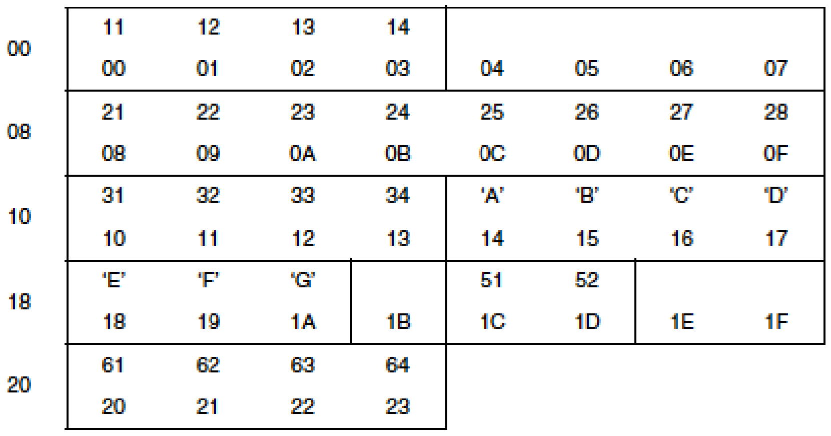
Figure 3-1. C Language Structure 's', Showing Values of Elements

```
struct {
     int        a;    /*          0x1112_1314word            */
     double     b;    /*          0x2122_2324_2526_2728doubleword */
     int        c;    /*          0x3132_3334 word           */
     char       d[7]; /*          'A', 'B', 'C', 'D', 'E', 'F', 'G' array of bytes */
     short      e;    /*          0x5152 halfword            */
     int        f;    /*          0x6162_6364 word           */
}s;
```

Big-Endian Mapping

C-language structure mapping rules permit the use of padding (skipped bytes) in order to align the scalars on desirable boundaries. This alignment introduces padding of four bytes between 'a' and 'b', one byte between 'd' and 'e', and two bytes between 'e' and 'f'. The consequent storage mapping of structure 's' is illustrated in *Figure 3-2*, which shows each scalar aligned at its natural boundary for big-endian mapping. Addresses are shown in hexadecimal at the left of each double word, and in figures below each byte. The contents of each byte, as indicated in *Figure 3-1*, are shown in hexadecimal (as characters for the elements of the string).

Figure 3-2. Big-Endian Mapping of Structure 's'



00	11	12	13	14				
	00	01	02	03	04	05	06	07
08	21	22	23	24	25	26	27	28
	08	09	0A	0B	0C	0D	0E	0F
10	31	32	33	34	'A'	'B'	'C'	'D'
	10	11	12	13	14	15	16	17
18	'E'	'F'	'G'		51	52		
	18	19	1A	1B	1C	1D	1E	1F
20	61	62	63	64				
	20	21	22	23				

3.2. Floating-Point and Vector/SIMD Multimedia Extension Data Format

Please refer to the following documents for more information about the Floating-Point and Vector/SIMD Multimedia Extension Data Format of PPE.

Floating-Point Data Format	"4.3 Floating-Point Data" of *Book I: PowerPC User Instruction Set Architecture, Version 2.02*
Vector/SIMD Multimedia Extension Data Format	"3 Operand Conventions" of *PowerPC Microprocessor Family: Vector/SIMD Multimedia Extension Technology Programming Environments Manual* Version 2.06c

4. Instruction Set Overview

Please refer to the following documents for more information about the Instruction Set of PPE.

Instruction Format	"1.7 Instruction Formats" of *Book I: PowerPC User Instruction Set Architecture, Version 2.02*
Branch Processor Instructions	"2.4 Branch Processor Instructions" of *Book I: PowerPC User Instruction Set Architecture, Version 2.02*
Fixed-Point Processor Instructions	"3.3 Fixed-Point Processor Instructions" of *Book I: PowerPC User Instruction Set Architecture, Version 2.02*
Floating-Point Processor Instructions	"4.6 Floating-Point Processor Instructions" of *Book I: PowerPC User Instruction Set Architecture, Version 2.02*
Vector/SIMD Multimedia Extension Instructions	*PowerPC Microprocessor Family: Vector/SIMD Multimedia Extension Technology Programming Environments Manual* Version 2.06c

5. Register Set

5.1. User and Vector/SIMD Multimedia Extension Register Set

Please refer to the following documents for more information about the User and Vector/SIMD Multimedia Extension register set of PPE.

User register set	"1.6 Processor Overview, Figure2. PowerPC user register set", "3.2 Fixed-Point Processor Registers", "2.3 Branch Processor Registers", and "4.2 Floating-Point Processor Registers" of *Book I: PowerPC User Instruction Set Architecture, Version 2.02*
Vector/SIMD Multimedia Extension Register	*PowerPC Microprocessor Family: Vector/SIMD Multimedia Extension Technology Programming Environments Manual* Version 2.06c

5.2. Privileged register

This section describes Special Purpose Registers (SPRs) used by the PowerPC Processor Element. To get the additional information about the privileged register of PPE, please refer to *Book III: PowerPC Operating Environment Architecture, Version 2.02*.

5.2.1. PowerPC Processor Element SPR Memory Map and Register Summary Table

Table 5-1 PowerPC Processor Element SPRs on page 14 contains a list of the Special Purpose Registers (SPRs) found in the PowerPC Processor Element. These registers are read from (mf) or written to (mt) using the **mfspr** and **mtspr** PowerPC Processor Element instructions respectively. *Table 5-1* uses the following conventions:

- Architected SPRs that do not contain implementation-specific information are included in the table, but are not described in this manual. The table provides cross-references to additional information for each architected SPR.
- Architected SPRs that contain implementation-specific information are included in the table and are described in this manual. When applicable, specific cross-references to additional information are provided in each register description. The SPRs in the *Decimal* column are highlighted to indicate when an SPR is implementation-specific.
- Notes are provided at the end of *Table 5-1* that describe unique and implementation-specific information for certain SPRs.

Please refer to *Appendix: SPR Definitions* for details.

Table 5-1. PowerPC Processor Element SPRs

SPR		Duplicated for multithreading [3]	Register Name (Short Name) *Cross Reference to Additional Information*	Unit	Read/Write	Synchronization Requirements [4]				Hypervisor/ Privileged [5]		Size (bits)	Power-On-Reset (POR) Value (All bits set to '0' unless otherwise noted)
						For Data		For Instructions					
Decimal [1]	spr[5:9] spr[0:4] [2]					Before Writes	After Writes	Before Writes	After Writes	Read (mf)	Write (mt)		
01	00000 00001	Yes	Fixed-Point Exception Register (XER) *Book I: PowerPC User Instruction Set Architecture, Version 2.02*	XU	R/W	N/A				—	—	64	
08	00000 01000	Yes	Link Register (LR) *Book I: PowerPC User Instruction Set Architecture, Version 2.02*	IU	R/W	N/A				—	—	64	
09	00000 01001	Yes	Count Register (CTR) *Book III: PowerPC Operating Environment Architecture, Version 2.02*	IU	R/W	N/A				—	—	64	
18	00000 10010	Yes	Data Storage Interrupt Status Register (DSISR) *Book III: PowerPC Operating Environment Architecture, Version 2.02*	XU	R/W	N/A				Priv		32	
19	00000 10011	Yes	Data Address Register (DAR) *Book III: PowerPC Operating Environment Architecture, Version 2.02*	XU	R/W	N/A				Priv		64	
22	00000 10110	Yes	Decrementer Register (DEC) *Book III: PowerPC Operating Environment Architecture, Version 2.02*	MMU	R/W	None				Priv		32	x'7FFF_FFFF'
25	00000 11001	No	Storage Description Register 1 (SDR1) *Book III: PowerPC Operating Environment Architecture, Version 2.02*	MMU	R/W	ptesync	CSI	ptesync	CSI	HV		64	
26	00000 11010	Yes	Machine Status Save/Restore Register 0 (SRR0) *Book III: PowerPC Operating Environment Architecture, Version 2.02*	IU	R/W	N/A				Priv		64	
27	00000 11011	Yes	Machine Status Save/Restore Register 1 (SRR1) *Book III: PowerPC Operating Environment Architecture, Version 2.02*	IU	R/W	N/A				Priv		64	
29	00000 11101	Yes	Address Compare Control Register (ACCR) *Book III: PowerPC Operating Environment Architecture, Version 2.02*	XU	R/W	CSI		None		Priv		64	
136	00100 01000	No	Control Register (CTRL) *Appendix: Control Register (CTRL)* on page 128	IU	R	N/A				—	N/A	32	
152	00100 11000				W	None				N/A	Priv[6]		N/A
256	01000 00000	Yes	VXU Register Save (VRSAVE) *12.1.3 Vector Multimedia Registers* on page 118 *PowerPC Microprocessor Family: Vector/SIMD Multimedia Extension Technology Programming Environments Manual Version 2.06c*	XU	R/W	N/A				—	—	32	
259	01000 00011	Yes	Software Use Special Purpose Register 3 (SPRG3) – Read Only *Book III: PowerPC Operating Environment Architecture, Version 2.02*	XU	R	N/A				—	N/A	64	

Table 5-1. PowerPC Processor Element SPRs

SPR		Duplicated for multithreading [3]	Register Name (Short Name) *Cross Reference to Additional Information*	Unit	Read/Write	Synchronization Requirements [4]				Hypervisor/ Privileged [5]		Size (bits)	Power-On-Reset (POR) Value (All bits set to '0' unless otherwise noted)
						For Data		For Instructions					
Decimal [1]	spr[5:9] spr[0:4] [2]					Before Writes	After Writes	Before Writes	After Writes	Read (mf)	Write (mt)		
268	01000 01100	No	Time Base Register (TB) *Book III: PowerPC Operating Environment Architecture, Version 2.02* The physical implementation of this register includes the following registers: *Time Base Register – Read Only (TB), SPR #268* *Time Base Upper Register – Read Only (TBU), SPR #269* *Time Base Lower Register – Write Only (TBL), SPR #284* *Time Base Upper Register – Write Only (TBU), SPR#285*	MMU	R	N/A				—	N/A	64	
269	01000 01101											32	
272	01000 10000	Yes	Software Use Special Purpose Register 0 (SPRG0) *Book III: PowerPC Operating Environment Architecture, Version 2.02*	XU	R/W	N/A				Priv		64	
273	01000 10001	Yes	Software Use Special Purpose Register 1 (SPRG1) *Book III: PowerPC Operating Environment Architecture, Version 2.02*	XU	R/W	N/A				Priv		64	
274	01000 10010	Yes	Software Use Special Purpose Register 2 (SPRG2) *Book III: PowerPC Operating Environment Architecture, Version 2.02*	XU	R/W	N/A				Priv		64	
275	01000 10011	Yes	Software Use Special Purpose Register 3 (SPRG3) *Book III: PowerPC Operating Environment Architecture, Version 2.02*	XU	R/W	N/A				Priv		64	
284	01000 11100	No	Time Base Register (TB) *Book III: PowerPC Operating Environment Architecture, Version 2.02* The physical implementation of this register includes the following registers: *Time Base Register – Read Only (TB), SPR #268* *Time Base Upper Register – Read Only (TBU), SPR #269* *Time Base Lower Register – Write Only (TBL), SPR #284* *Time Base Upper Register – Write Only (TBU), SPR#285*	MMU	W	None				N/A	HV	32	N/A
285	01000 11101											32	
287	01000 11111	No	PPE Processor Version Register (PVR) *Appendix: PPE Processor Version Register (PVR)* on page 129	XU	R	N/A				Priv	N/A	32	x'0070_0100' (DD1) x'0070_0400' (DD2) x'0070_0501' (DD3.1)
304	01001 10000	Yes	Hypervisor Software Use Special Purpose Register 0 (HSPRG0) [7] *Book III: PowerPC Operating Environment Architecture, Version 2.02*	XU	R/W	N/A				HV		64	
305	01001 10001	Yes	Hypervisor Software Use Special Purpose Register 1 (HSPRG1) [7] *Book III: PowerPC Operating Environment Architecture, Version 2.02*	XU	R/W	N/A				HV		64	
309	01001 10101		Processor Utilization of Resources Register (PURR) *Book III: PowerPC Operating Environment Architecture, Version 2.02*		R/W					HV		64	
310	01001 10110	No	Hypervisor Decrementer Register (HDEC) [7] *Book III: PowerPC Operating Environment Architecture, Version 2.02*	MMU	R/W	None				HV		32	x'7FFF_FFFF'

Table 5-1. PowerPC Processor Element SPRs

SPR		Duplicated for multithreading [3]	Register Name (Short Name) *Cross Reference to Additional Information*	Unit	Read/Write	Synchronization Requirements [4]				Hypervisor/ Privileged [5]		Size (bits)	Power-On-Reset (POR) Value (All bits set to '0' unless otherwise noted)
						For Data		For Instructions					
Decimal [1]	spr[5:9] spr[0:4] [2]					Before Writes	After Writes	Before Writes	After Writes	Read (mf)	Write (mt)		
312	01001 11000	No	Real Mode Offset Register (RMOR) *Appendix: Real Mode Offset Register (RMOR)* on page 130	MMU	R/W	CSI		None	CSI	HV		64	
313	01001 11001	No	Hypervisor Real Mode Offset Register (HRMOR) [7] Appendix: *Hypervisor Real Mode Offset Register (HRMOR)* on page 130	MMU	R/W	CSI		None	CSI	HV		64	
314	01001 11010	Yes	Hypervisor Machine Status Save/Restore Register 0 (HSRR0) [7] *Book III: PowerPC Operating Environment Architecture, Version 2.02*	IU	R/W	N/A				HV		64	
315	01001 11011	Yes	Hypervisor Machine Status Save/Restore Register 1 (HSRR1) [7] *Book III: PowerPC Operating Environment Architecture, Version 2.02*	IU	R/W	N/A				HV		64	
318	01001 11110	Partial	Logical Partition Control Register (LPCR) [7] *Appendix: Logical Partition Control Register (LPCR)* on page 131	MMU	R/W	CSI		None	CSI	HV		64	
319	01001 11111	No	Logical Partition Identity Register (LPIDR) [7] *Appendix: Logical Partition Identity Register (LPIDR)* on page 132	MMU	R/W	CSI		CSI		HV		32	
896	11100 00000	Yes	Thread Status Register Local (TSRL) *Appendix: Thread Status Register Local (TSRL)* on page 133	IU	R/W	None			CSI	—	—	64	
897	11100 00001	Yes	Thread Status Register Remote (TSRR) *Appendix: Thread Status Register Remote (TSRR)* on page 134	IU	R	N/A				—	N/A	64	
921	11100 11001	No	Thread Switch Control Register (TSCR) *Appendix: Thread Switch Control Register (TSCR)* on page 135	IU	R/W	None			CSI	HV		64	
922	11100 11010	No	Thread Switch Time-Out Register (TTR) *Appendix: Thread Switch Time-Out Register (TTR)* on page 136	IU	R/W	None			CSI	HV		64	
946	11101 10010	Yes	PPE Translation Lookaside Buffer Index Hint Register (PPE_TLB_Index_Hint) *Appendix: PPE Translation Lookaside Buffer Index Hint Register (PPE_TLB_Index_Hint)* on page 137	MMU	R	N/A				Priv	N/A	64	
947	11101 10011	No	PPE Translation-Lookaside Buffer Index Register (PPE_TLB_Index) *Appendix: PPE Translation-Lookaside Buffer Index Register (PPE_TLB_Index)* on page 138	MMU	R/W [8]	None				HV		64	
948	11101 10100	No	PPE Translation-Lookaside Buffer Virtual-Page Number Register (PPE_TLB_VPN) *Appendix: PPE Translation-Lookaside Buffer Virtual-Page Number Register (PPE_TLB_VPN)* on page 139	MMU	R/W	CSI	CSI	None	CSI	HV		64	
949	11101 10101	No	PPE Translation-Lookaside Buffer Real-Page Number Register (PPE_TLB_RPN) *Appendix: PPE Translation-Lookaside Buffer Real-Page Number Register (PPE_TLB_RPN)* on page 140	MMU	R/W	None				HV		64	

Table 5-1. PowerPC Processor Element SPRs

SPR		Duplicated for multithreading [3]	Register Name (Short Name) *Cross Reference to Additional Information*	Unit	Read/Write	Synchronization Requirements [4]				Hypervisor/ Privileged [5]		Size (bits)	Power-On-Reset (POR) Value (All bits set to '0' unless otherwise noted)
						For Data		For Instructions					
Decimal [1]	spr[5:9] spr[0:4] [2]					Before Writes	After Writes	Before Writes	After Writes	Read (mf)	Write (mt)		
951	11101 10111	No	PPE Translation-Lookaside Buffer RMT Register (PPE_TLB_RMT) *Appendix: PPE Translation-Lookaside Buffer RMT Register (PPE_TLB_RMT)* on page 141	MMU	R/W	CSI	CSI	None	CSI	HV		64	
952	11101 11000	Yes	Data Address Range Start Register 0 (DRSR0) *Cell Broadband Engine™ Architecture* *Appendix: Data Address Range Start Register 0 (DRSR0)* on page 142	XU	R/W	Sync	Sync, CSI[9]	None	None	HV		64	
953	11101 11001	Yes	Data Range Mask Register 0 (DRMR0) *Cell Broadband Engine™ Architecture* *Appendix: Data Range Mask Register 0 (DRMR0)* on page 142	XU	R/W	Sync	Sync, CSI[9]	None	None	HV		64	
954	11101 11010	Yes	Data Class ID Register 0 (DCIDR0) *Cell Broadband Engine™ Architecture* *Appendix: Data Class ID Register 0 (DCIDR0)* on page 142	XU	R/W	Sync	Sync, CSI[9]	None	None	HV		32	
955	11101 11011	Yes	Data Range Start Register 1 (DRSR1) *Cell Broadband Engine™ Architecture* *Appendix: Data Range Start Register 1 (DRSR1)* on page 143	XU	R/W	Sync	Sync, CSI[9]	None	None	HV		64	
956	11101 11100	Yes	Data Range Mask Register 1 (DRMR1) *Cell Broadband Engine™ Architecture* *Appendix: Data Range Mask Register 1 (DRMR1)* on page 143	XU	R/W	Sync	Sync, CSI[9]	None	None	HV		64	
957	11101 11101	Yes	Data Class ID Register 1 (DCIDR1) *Cell Broadband Engine™ Architecture* *Appendix: Data Class ID Register 1 (DCIDR1)* on page 143	XU	R/W	Sync	Sync, CSI[9]	None	None	HV		32	
976	11110 10000	Yes	Instruction Range Start Register 0 (IRSR0) *Cell Broadband Engine™ Architecture* *Appendix: Instruction Range Start Register 0 (IRSR0)* on page 144	IU	R/W	None	None	Sync	Sync, CSI[9]	HV		64	
977	11110 10001	Yes	Instruction Range Mask Register 0 (IRMR0) *Cell Broadband Engine™ Architecture* *Appendix: Instruction Range Mask Register 0 (IRMR0)* on page 144	IU	R/W	None	None	Sync	Sync, CSI[9]	HV		64	
978	11110 10010	Yes	Instruction Class ID Register 0 (ICIDR0) *Cell Broadband Engine™ Architecture* *Appendix: Instruction Class ID Register 0 (ICIDR0)* on page 144	IU	R/W	None	None	Sync	Sync, CSI[9]	HV		32	
979	11110 10011	Yes	Instruction Range Start Register 1 (IRSR1) *Cell Broadband Engine™ Architecture* *Appendix: Instruction Range Start Register 1 (IRSR1)* on page 145	IU	R/W	None	None	Sync	Sync, CSI[9]	HV		64	

Table 5-1. PowerPC Processor Element SPRs

SPR		Duplicated for multithreading [3]	Register Name (Short Name) *Cross Reference to Additional Information*	Unit	Read/Write	Synchronization Requirements [4]				Hypervisor/ Privileged [5]		Size (bits)	Power-On-Reset (POR) Value (All bits set to '0' unless otherwise noted)
Decimal [1]	spr[5:9] spr[0:4] [2]					For Data		For Instructions					
						Before Writes	After Writes	Before Writes	After Writes	Read (mf)	Write (mt)		
980	11110 10100	Yes	Instruction Range Mask Register 1 (IRMR1) *Cell Broadband Engine™ Architecture* *Appendix: Instruction Range Mask Register 1 (IRMR1)* on page 145	IU	R/W	None	None	Sync	Sync, CSI[9]	HV		64	
981	11110 10101	Yes	Instruction Class ID Register 1 (ICIDR1) *Cell Broadband Engine™ Architecture* *Appendix: Instruction Class ID Register 1 (ICIDR1)* on page 145	IU	R/W	None	None	Sync	Sync, CSI[9]	HV		32	
1008	11111 10000	No	Hardware Implementation Register 0 (HID0) *Appendix: Hardware Implementation Register 0 (HID0)* on page 146	IU	R/W	Sync	Sync, CSI[9]	Sync	Sync, CSI[9]	HV		64	
1009	11111 10001	No	Hardware Implementation Register 1 (HID1) *Appendix: Hardware Implementation Register 1 (HID1)* on page 148	IU	R/W	Sync	Sync, CSI[9]	Sync	Sync, CSI[9]	HV		64	
1012	11111 10100	No	Hardware Implementation Register 4 (HID4) *Appendix: Hardware Implementation Register 4 (HID4)* on page 151	XU	R/W	Sync	Sync, CSI[9]	Sync	Sync, CSI[9]	HV		64	
1013	11111 10101	Yes	Data Address Breakpoint Register (DABR) *Book III: PowerPC Operating Environment Architecture, Version 2.02*	XU	R/W	Sync	CSI	None		HV		64	
1014	11111 10110	No	Hardware Implementation Register 5 (HID5) *Appendix: Hardware Implementation Register 5 (HID5)* on page 153	XU	R/W	Sync	Sync, CSI[9]	Sync	Sync, CSI[9]	HV		64	
1015	11111 10111	Yes	Data Address Breakpoint Register Extension (DABRX) *Book III: PowerPC Operating Environment Architecture, Version 2.02*	XU	R/W	Sync	CSI	None		HV		64	
1016	11111 11000	Yes	Trigger Data Address Breakpoint Register Extension (TDABRX) *For hardware-debug purpose only*	XU	R/W	Sync	CSI	None		Priv		64	
1017	11111 11001	No	Hardware Implementation Register 6 (HID6) *Appendix: Hardware Implementation Register 6 (HID6)* on page 154	MMU	R/W	Sync	Sync, CSI[9]	Sync	Sync, CSI[9]	HV		64	
1018	11111 11010	No	Hardware Implementation Register 7 (HID7) *Appendix: Hardware Implementation Register 7 (HID7)* on page 155	IU	R/W	Sync	Sync, CSI[9]	Sync	Sync, CSI[9]	HV		64	
1019	11111 11011	Yes	Trigger Instruction Address Breakpoint Register (TIABR) *For hardware-debug purpose only*	IU	R/W	None		Sync	CSI	Priv		64	
1020	11111 11100	No	Bookmark (BKMK) *For hardware-debug purpose only*	IU	W	N/A				N/A	Priv	64	
1021	11111 11101	Yes	Trigger Data Address Breakpoint Register (TDABR) *For hardware-debug purpose only*	XU	R/W	CSI	CSI	None		Priv		64	

Table 5-1. PowerPC Processor Element SPRs

SPR		Duplicated for multithreading [3]	Register Name (Short Name) *Cross Reference to Additional Information*	Unit	Read/Write	Synchronization Requirements [4]				Hypervisor/ Privileged [5]		Size (bits)	Power-On-Reset (POR) Value (All bits set to '0' unless otherwise noted)
						For Data		For Instructions					
Decimal [1]	spr[5:9] spr[0:4] [2]					Before Writes	After Writes	Before Writes	After Writes	Read (mf)	Write (mt)		
1022	11111 11110	No	CBEA-Compliant Processor Version Register (BP_VR) *Appendix: CBEA-Compliant Processor Version Register (BP_VR)* on page 129 *Cell Broadband Engine™ Architecture*	XU	R	N/A				Priv	N/A	32	See page 129
1023	11111 11111	Yes	Processor Identification Register (PIR) *Appendix: Processor Identification Register (PIR)* on page 129	XU	R	N/A				Priv	N/A	32	See page 129

1. SPRs with implementation-specific settings are highlighted.
2. The order of the two five-bit halves of the SPR number is reversed (to follow the convention of the architecture documents).
3. Any register that is non-duplicated per thread requires special care by the Hypervisor when written. The Hypervisor must not cause an implicit branch or undefined behavior for the thread that is not writing the register.
4. For more information, see the *Synchronization Requirements for Context Alterations* section of *Book III, PowerPC Operating Environment Architecture, Version 2.02*. In this column, Sync refers to the lightweight **sync** L=1 instruction unless otherwise indicated.
5. Explanation of the *Hypervisor/Privileged* column:
   HV: indicates that the register is a hypervisor resource and the hypervisor state must be enabled (MSR[HV]='1') to write this register. Attempts to modify the contents of a hypervisor resource (such as using the move-to spr instruction) in privileged, but non-hypervisor state (MSR[HV, PR]='00'), cause a privileged-instruction program interrupt.
   Priv: indicates that the register is privileged, and the privileged state must be enabled (MSR[PR]='0') to read or write to the register. Attempts to access the contents of a privileged resource (such as using the move-to spr or move-from spr instruction) in non-privileged state (MSR[PR]='1'), cause a privileged-instruction program interrupt.
   — : indicates that the register is neither privileged nor a hypervisor resource.
   N/A: indicates that the register is either read-only or write-only.
   – Writes using move-to instructions to unimplemented SPRs are treated as nops. Architected registers are not changed.
   – Writes using move-to instructions to read-only SPRs are treated like unimplemented SPRs.
   – Reads using move-from instructions from unimplemented SPRs cause zeros to be written back to the GPR.
   – Reads using move-from instructions from write-only SPRs are treated like unimplemented SPRs.
6. The Thread Enable Bits field of CTRL can be modified only by hypervisor mode. Attempts to modify (using the move-to spr instruction) the Thread Enable Bits field of the CTRL register while not in hypervisor state (MSR[HV]='0') are ignored.
7. These registers are for LPAR support.
8. Reading of these registers is allowed for diagnostic purposes.
9. Indicates a **sync** instruction followed by any context synchronization instruction.

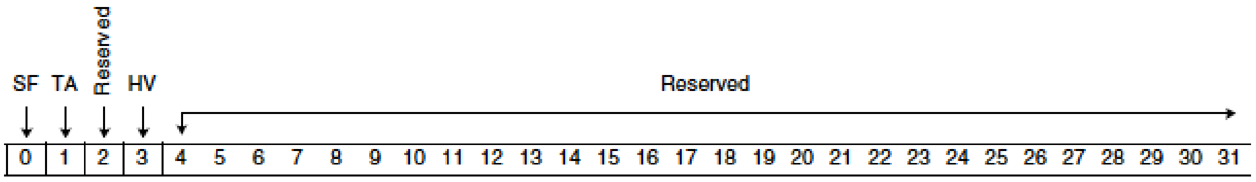
5.2.2. Special Architected Registers

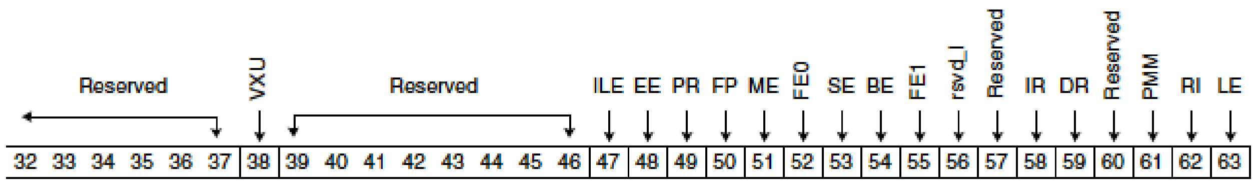
This section describes registers that have been architected to meet special requirements. These registers may have unique characteristics, uses, and applications.

5.2.2.1. Machine State Register (MSR)

Register Short Name	MSR	**Unit**	IU
Register Type	SPR Read/Write	**Width**	64 bits
Decimal SPR Number	N/A	**Access Type**	Read: Privileged Write: Privileged
Value at Initial POR[1]	All bits set to zero	**Set By**	Scan initialization during POR
Additional Information	PowerPC architected register	**Reg. Duplicated for MT**	Yes

1. A system reset sets the POR value to the value indicated by system reset interrupt which is x'9000_0000_0000_0000' (Where the 9 means SF = '1' and HV = '1').





Bit(s)	Field Name	Description
0	SF	Sixty-four bit mode 0 The processor is in 32-bit mode. 1 The processor is in 64-bit mode.
1	TA	Tags-active mode This mode is not supported. This bit is reserved and is forced to zero.
2	Reserved	Bits are not implemented; all bits read back zero.
3	HV	Hypervisor State 0 The processor is not in hypervisor state. 1 If MSR[PR] is set to '0', the processor is in hypervisor state; otherwise, the processor is in problem state.
4:37	Reserved	Bits are not implemented; all bits read back zero.
38	VXU	VXU 0 The processor cannot execute VXU instructions. If the processor attempts to execute a VXU instruction, this causes a VXU Unavailable interrupt. 1 The processor can execute VXU instructions.
39:46	Reserved	Bits are not implemented; all bits read back zero.
47	ILE	Interrupt Little-Endian mode This mode is not supported. This bit is reserved and forced to '0'.
48	EE	External interrupt enable 0 External and Decrementer interrupts are disabled. 1 External and Decrementer interrupts are enabled.
49	PR	Problem state 0 The processor is in privileged state. 1 The processor is in problem state.

Bit(s)	Field Name	Description
50	FP	Floating-point available 0 The processor cannot execute any floating-point instructions, including floating-point loads, stores, and moves. 1 The processor can execute floating-point instructions.
51	ME	Machine check enable 0 Machine Check interrupts are disabled. 1 Machine Check interrupts are enabled. This bit is a hypervisor resource; see *Book III: PowerPC Operating Environment Architecture, Version 2.02*.
52	FE0	Floating point exception 0 The PPE does not support imprecise-nonrecoverable mode nor imprecise-recoverable mode. The Floating-Point Exception Mode bits FE0 and FE1 are interpreted as shown below: [FE0] [FE1] Mode 0 0 Ignore exceptions mode 0 1 Precise mode 1 0 Precise mode 1 1 Precise mode (For additional information about floating point exception modes, *Section 11.3* Floating-Point Exception Mode on page 110.)
53	SE	Single-step trace enable 0 The processor executes instructions normally. 1 The processor generates a single-step type trace interrupt after successfully completing the execution of the next instruction (unless that instruction is **rfid**, **hrfid**, **attn**, or **sc** which are never traced). Successful completion signifies that the instruction caused no other interrupt.
54	BE	Branch trace enable 0 The processor executes branch instructions normally. 1 The processor generates a branch type trace interrupt after completing the execution of a branch instruction, whether or not the branch is taken.
55	FE1	Floating point exception 1 **Note:** See description of the *FE0* bit in this register
56	rsvd_l	Reserved. Latch bit is implemented; value read is the value written.
57	Reserved	Bits are not implemented; all bits read back zero.
58	IR	Instruction relocate 0 Instruction address translation is off. 1 Instruction address translation is on.
59	DR	Data relocate 0 Data address translation is off 1 Data address translation is on
60	Reserved	Bits are not implemented; all bits read back zero.
61	PMM	Performance Monitor Mark This bit is reserved and forced to zero.
62	RI	Recoverable Interrupt 0 Interrupt is not recoverable 1 Interrupt is recoverable
63	LE	Little-endian mode This mode is not supported. This bit is reserved and is forced to '0', which means the mode is always Big-Endian.

Additional Information:

- *Book III: PowerPC Operating Environment Architecture, Version 2.02*

6. PowerPC Processor Unit

6.1. PowerPC Processor Unit (PPU) Overview

The PowerPC Processor Unit (PPU) is a dual-threaded processing unit responsible for overall control of the system. It runs the operating systems and provides flow control for the various applications running on the chip. *Figure 6-1 PowerPC Processor Unit High Level Block Diagram* on page 23 contains a high level block diagram of the PPU.

The PPU consists of the Instruction Unit (IU); the Execution Unit (XU), which is further divided into the Fixed-Point Integer Unit (FXU) and the Load and Store Unit (LSU); the Memory Management Unit (MMU); and the Vector/Scalar Unit (VSU), which is further divided into the double-precision Floating-Point Unit (FPU) and Vector/SIMD Multimedia Extension Unit (VXU).

6.1.1. Instruction Unit (IU)

The instruction unit (IU) has the following implemented features:

- A 32 KB, two-way set-associative, L1 Instruction Cache (ICache) with a 128 B cache-line size
- A 64 entry two-way set-associative instruction effective-to-real-address translation (I-ERAT) cache
- A four instruction-per-cycle fetcher, duplicated for each thread
- A four instruction wide by five entry deep instruction buffer (Ibuf), duplicated for each thread
- A 4 KB x 2 Branch History Table (BHT) with 6 bits of global history, duplicated for each thread
- A ROM for microcode execution of complex instructions, shared by both threads
- Separate VXU/FPU and LSU/FXU/BRU dual-issue points
- Out-of-order completion of load cache-miss instructions (otherwise in-order execution)
- A 12 stage VSU Issue Queue for separate handling of VXU/FPU Instructions

6.1.2. Fixed Point and Load/Store Execution Unit (XU)

The fixed-point and load/store-execution unit (XU) has the following implemented features:

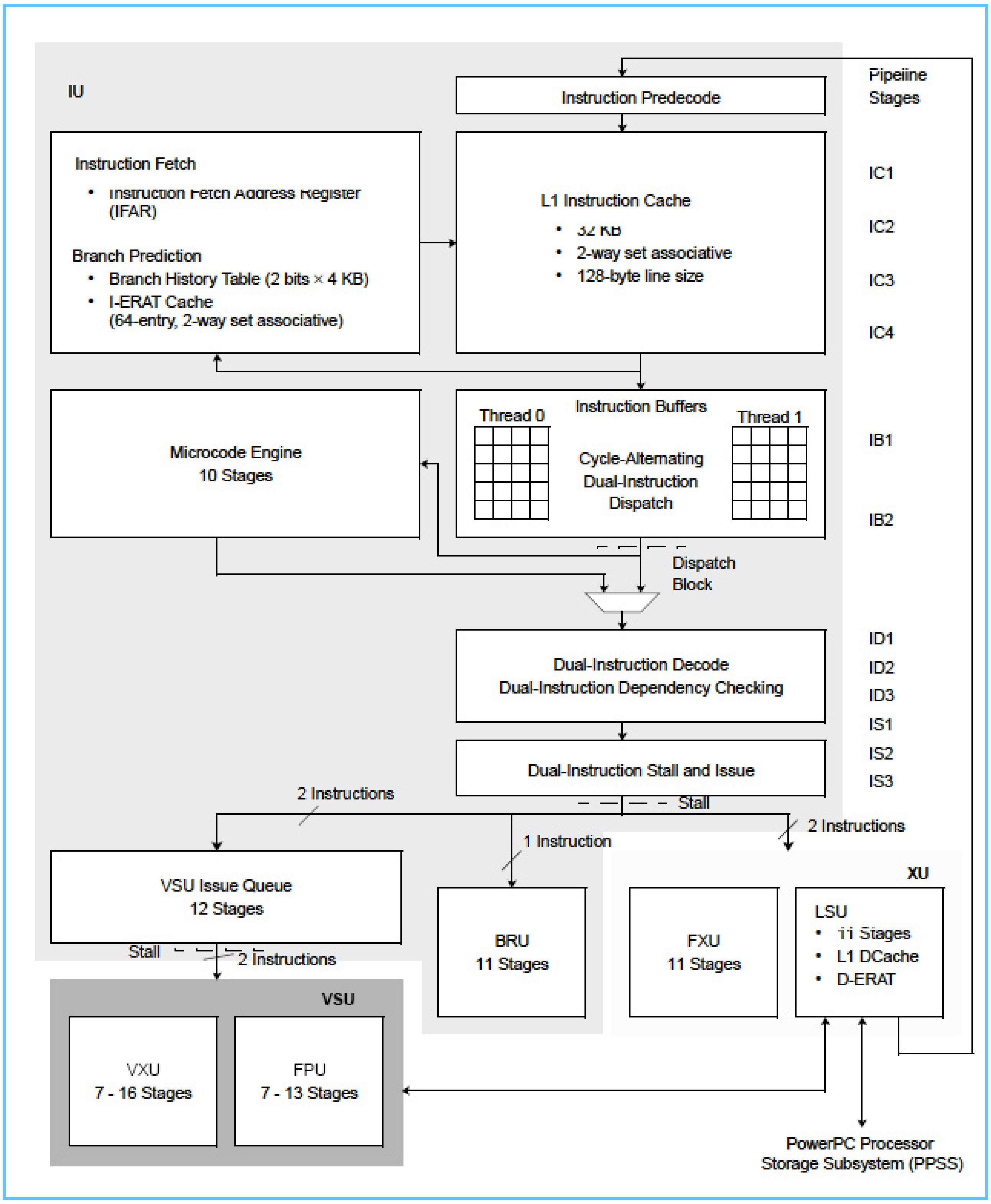
- A 32 KB, four-way set-associative, 128 B Line-Size, Write-Through L1 Data Cache (DCache)
- A 64-entry two-way set-associative data effective-to-real-address translation (D-ERAT) cache
- An eight-entry load-miss queue (shared between threads)
- A sixteen-entry store queue (shared between threads)
- A three-cycle delayed FXU to prevent load-dependent stalls on fixed-point instructions
- A five-read port and two-write port (5R/2W) General Purpose Register (GPR) file duplicated per thread
- Supports back-invalidation for L1 DCache snooping
- Supports a nonblocking L1 DCache (hit under miss)
- For MMU features see *Section 6.4.3* on page 39

6.1.3. Vector/Scalar Unit (VSU)

The VXU and FPU units are collectively referred to as the VSU (Vector-Scalar Unit).

The VSU executes all Vector/SIMD Multimedia Extension and Floating Point instructions that are not loads or stores. Vector/SIMD Multimedia Extension instructions are described in *PowerPC Microprocessor Family: Vector/SIMD Multimedia Extension Technology Programming Environments Manual* Version 2.06c and FPU instructions are described in *Chapter 4* of *Book I: PowerPC User Instruction Set Architecture*. Architected resources include the Vector Multimedia Register (VMR), Vector Status and Control Register (VSCR), the Floating-Point Register (FPR), and the Floating-Point Status and Control Register (FPSCR).

Figure 6-1. PowerPC Processor Unit High Level Block Diagram



6.2. PPU Pipeline

Figure 6-2 and *Figure 6-3* show the PPU pipeline.
The pipeline frontend consists of a ICache fetch, which is four cycles followed by a two-cycle Dispatch, a three-cycle Decode and a three-cycle Issue. VSU (VXU or FPU) instructions are issued to a separate VSU issue queue, which has a separate dependency-checking mechanism.

The execute end of the pipeline consists of a two-cycle register file access followed by multiple execute stages. Simple FXU operations execute in two cycles. A load takes five cycles. FPU double precision instructions take ten cycles. All fixed point instructions write back in the same cycle, after the final exception point.

The pipeline contains three stall points at IB2, IS2 and VSU Issue Queue. These stall points are primarily activated by operand dependency checking. Once an instruction is past all stall points it must either flow to completion or be flushed.

Figure 6-2. PPU IU, FXU, and LSU Pipeline

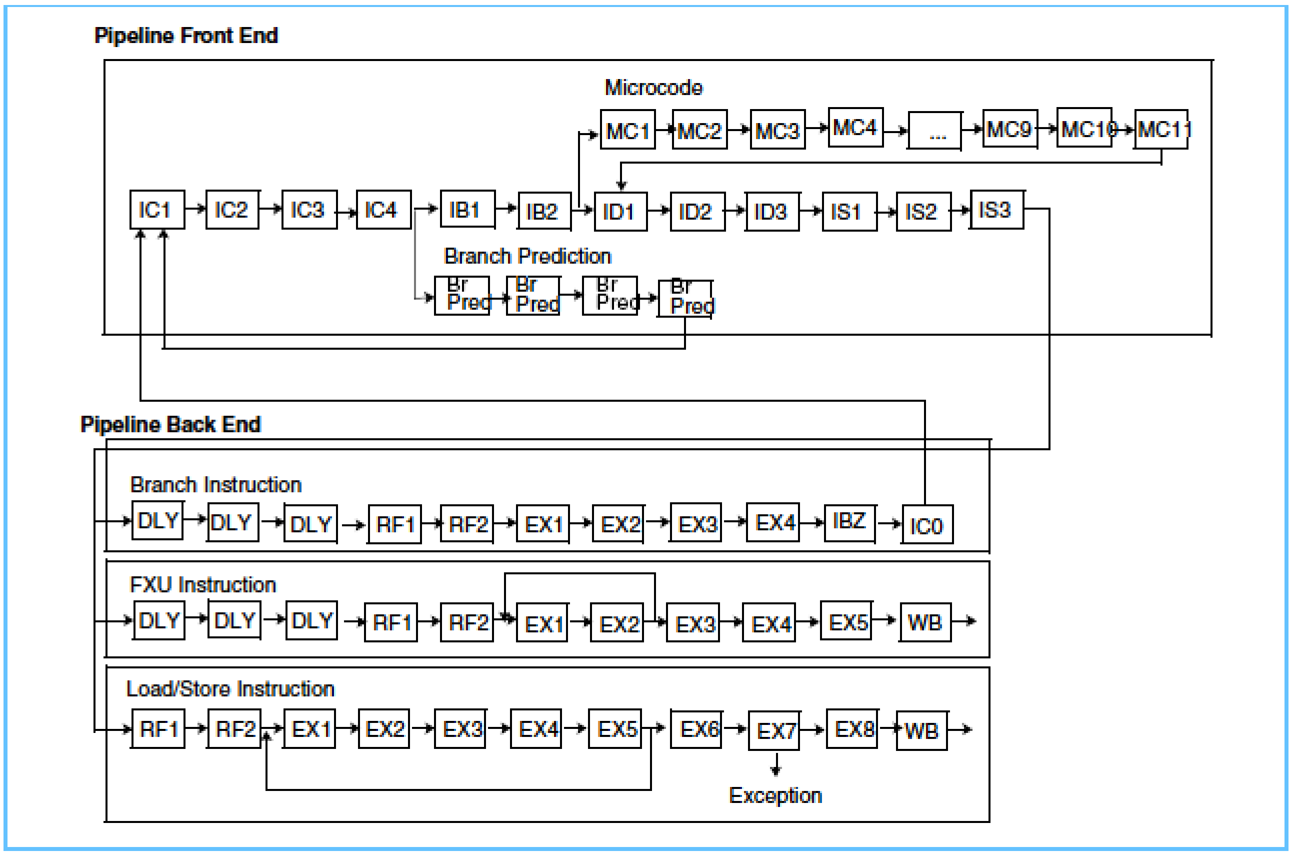
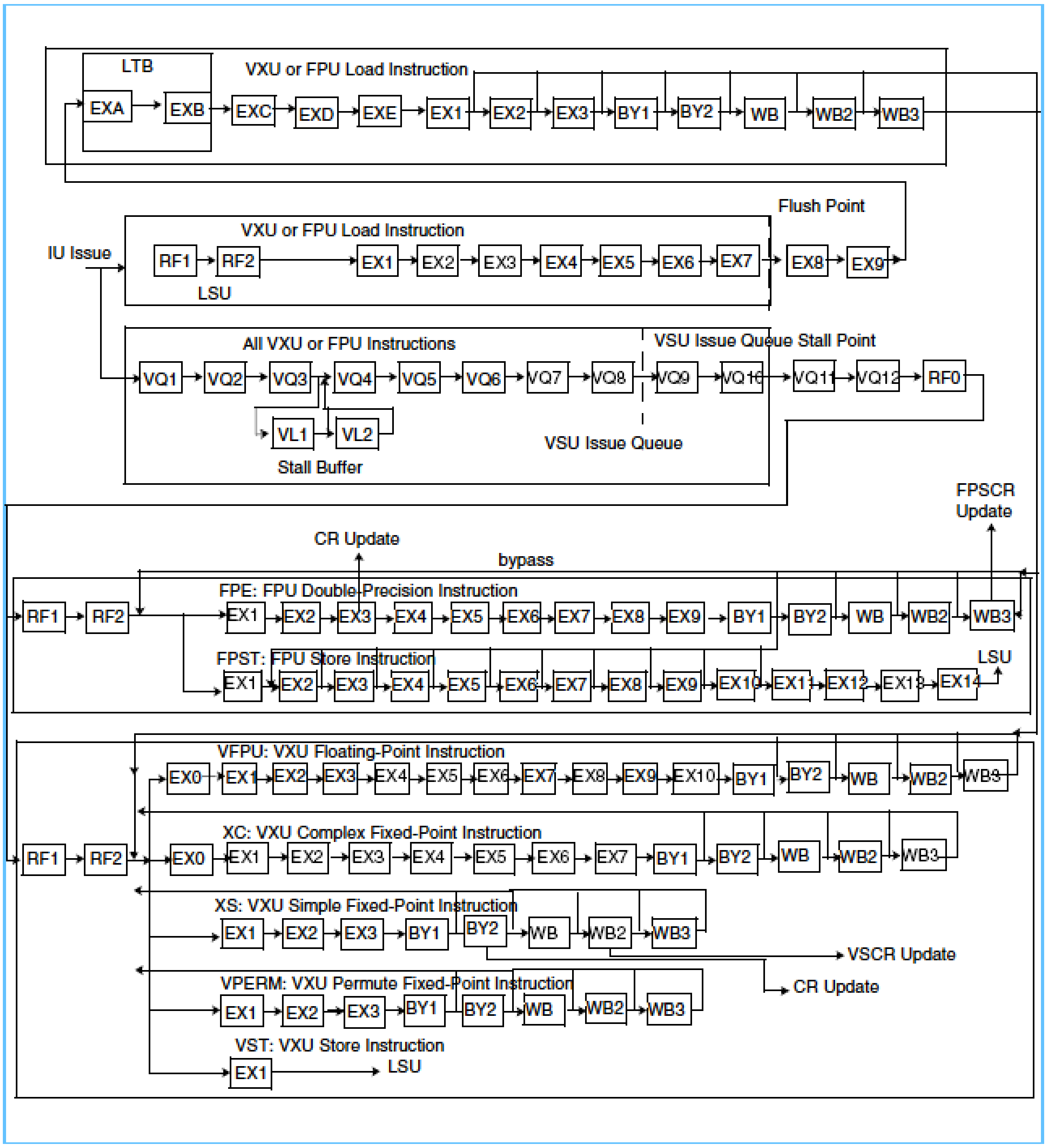


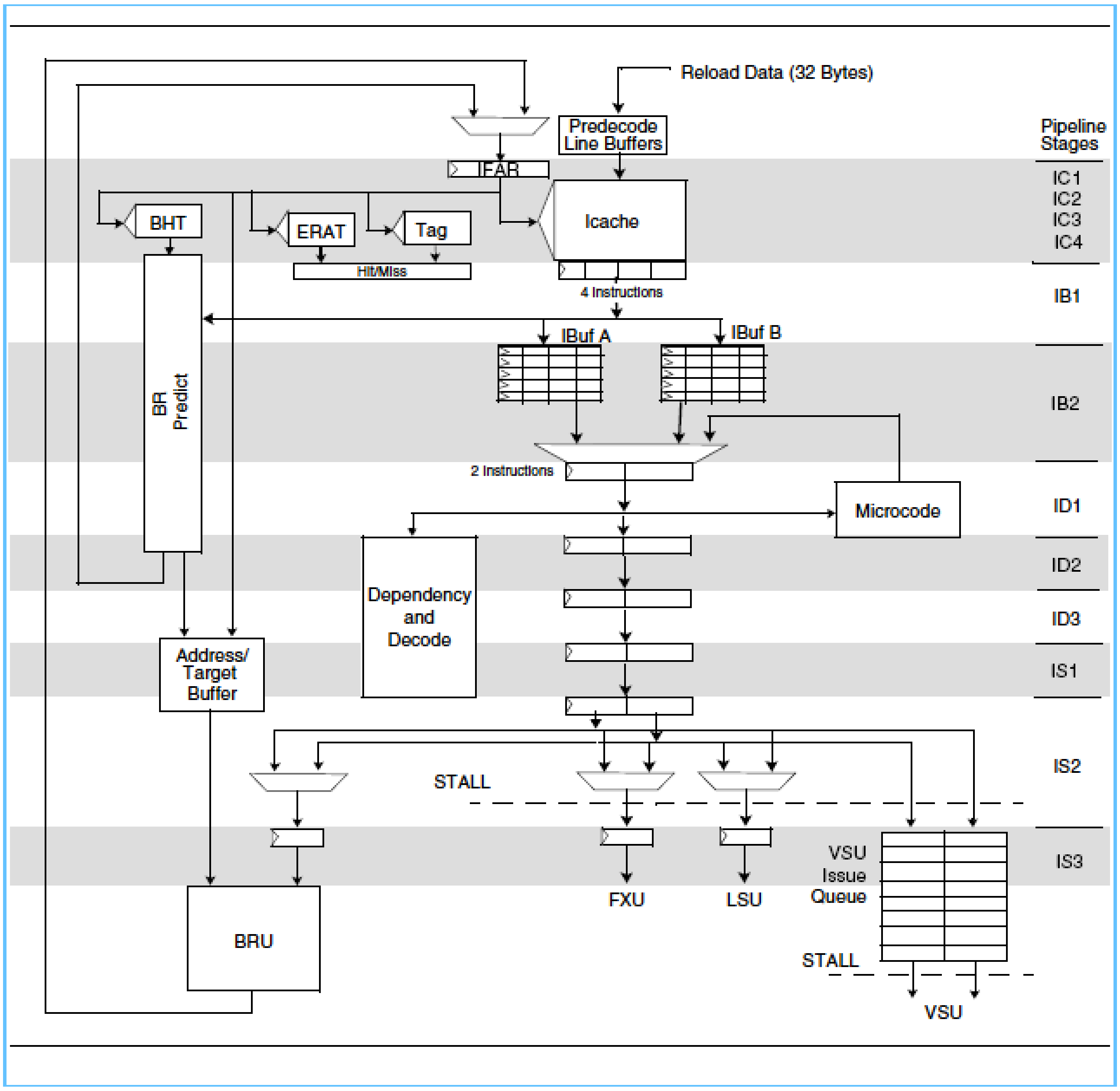
Figure 6-3. Floating Point and VXU Pipeline



6.3. Instruction Unit (IU)

Figure 6-4 IU Dataflow Diagram shows a high level block diagram of the IU. The IU is responsible for all instruction handling and control. After instructions are issued from the IU, they enter one of the execution units (LSU, FXU, VXU, FPU, or BRU). The BRU is an execution unit, but is considered part of the IU since it deals with instruction branching.

Figure 6-4. IU Dataflow Diagram



6.3.1. IU Pipeline

The front end of the IU pipeline consists of a 4-cycle cache access: IC1 through IC4. The fetch address is based on either the sequential path or the branch target path. In addition, the branch execution stage can correct the instruction fetch with the branch target address if the branch is mispredicted not-taken and with the sequential address if the branch is mispredicted taken.

After pipestage IC4, there are 2 instruction buffer stages: IB1 and IB2. The instruction buffer (Ibuf) is a first-in-first-out (FIFO) queue that is used to buffer up the 4 instructions fetched from the L1 ICache when there is a downstream stall condition, as discussed below. Instruction buffer stage IB1 is used to load the IBuf; there is one set of IBufs for each thread. IB2 is used to unload the IBuf and multiplex down to two instructions (this is the instruction dispatch stage). Each thread is given equal priority in dispatch, toggling every other cycle, unless specified otherwise in software (see *Section 6.6.3.2 Thread Priority* on page 50). If one thread is not fetching instructions into the IBuf (due to conditions such as a cache miss or thread disabled), then the other thread may have exclusive access to dispatch. Dispatch also controls the flow of instructions to and from microcode engine, which is used to break a PowerPC Architecture operation (instruction) that is difficult to execute into multiple smaller operations ("micro-ops").

Dispatch grouping occurs based on the effective address bits [60:63] of the instruction. The dispatcher groups instructions with addresses of x’0’ and x’4’, as well as instructions with addresses of x’8’ and x’C’. An instruction with address x‘0’ cannot be grouped with an instruction with address x‘8’ or x‘C’. Similarly, an instruction with address x‘4’ cannot be grouped with an instruction with address x‘8’ or x‘C’. In other words, instructions grouped together must be aligned on 8-byte boundaries.

Decode pipestages ID1 through ID3 are used to assemble the instruction internal opcodes and register source/target fields. In addition, dependency checking starts in ID2, which checks for data hazards (such as read-after-write or write-after-write). The issue logic continues in ID3, IS1, and IS2 to create a single stall point at IS2, which is propagated up the pipeline to the IBufs, stalling both threads. The IS2 stall point is driven by data-hazard detection, in addition to resource-conflict detections, among other conditions. The IS2 issue stage determines the appropriate routing of the instructions, upon which they are issued to the execution units in the IS3 cycle. Each instruction in IS2 can be routed to 5 different issue slots: to the fixed-point unit, load-store unit, branch unit, and to two slots in the VSU issue queue.

VSU instructions are issued to a separate VSU issue queue (VIQ), without checking for dependencies or issue restrictions. The VIQ has a separate dependency-checking facility, used exclusively for VSU instructions. A separate stall point is generated at the bottom of the VIQ in cycle VQ8.

6.3.2. Stall & Flush Points

In normal instruction execution, everything flows in an orderly pipelined fashion. Several conditions exist where this normal execution is interrupted with a stall or a flush condition. There are three separate stall points in the PPU.

The first stall point occurs just after the Instruction Buffers in the IB2 stage. This stall point is commonly referred to as a “dispatch block” because it prevents instructions from being dispatched. There are several possible reasons for a dispatch block (for more information, see *Section 6.6.3.1* Stalling on page 48). The following are two of the more common reasons:

1) A special **nop** instruction (see *Section 11.5 Nop Forms of the OR/ORI Instructions* on page 110)
2) The flush logic whenever an L1 Data Cache miss dependency (Read-After-Write (RAW) and Write-After-Write (WAW)) or D-ERAT miss occurs; or a Condition Register Field 0 (CR0) source-dependent operation issues while a **stdcx.** or **stwcx.** instruction is pending.

The benefit to stalling at this point in the pipeline is that it allows the other thread to continue dispatching instructions while the current thread is stalled. All other stall points in the pipeline block both threads. For further details, see *Section 6.6.3.1 Stalling* on page 48.

The second stall point occurs in the Issue logic at the IS2 stage. This stall point is activated by the hardware whenever one of the following conditions is met:

- An LSU or FXU instruction dependency occurs.
- A nonpipelined instruction is issued.
- An invalid dual-issue combination is present.
- A context serializing instruction is issued.
- The processor is in single-stepping mode.
- The lower VSU stall point (in cycle VQ8) is active and a VSU instruction is presently trying to issue to the VSU Issue Queue.

This stall point is also activated whenever a stall request is received from the LSU. This stall request is always honored if there is a valid instruction at IS2. A stall at IS2 also causes the microcode pipeline to stall.

The third stall point is exclusive to the VSU Issue Queue (in cycle VQ8) and is activated for the following conditions:

- A VXU or FPU dependency occurs.
- VXU or FPU write-port conflicts exist on the Vector Multimedia Register (VMR) or Floating-Point Register (FPR).
- An invalid dual-issue combination is present.
- A nonpipelined instruction is issued.
- There are special single-stepping conditions.

This stall point is also activated whenever a stall request is received from the LSU; this stall request is always honored.

A two-stage buffer separates the third stall point from the second stall point to help decouple the FXU and LSU instruction flow from the VXU and FPU instruction flow. If this buffer fills up and there is an instruction in the IS2 stage that needs to issue to the VSU Issue Queue, then the second stall point activates to prevent overflowing the VSU Issue Queue.

Once an instruction is past all stall points, it must either flow to completion or be flushed. The flush is initiated by one of the execution units in the back-end of the pipeline for various reasons such as the occurrence of a dependency on a cache miss or an exception condition. Whenever a flush occurs because of an exception, an L1 Data Cache miss dependency, or a D-ERAT address translation miss, it is taken at the ninth stage of the execution unit (EX7). All instructions in the machine that have been fetched from the L1 Instruction Cache and are younger (in program order) than the flushing instruction are invalidated.

The VXU and FPU contain an "internal" flush for instructions that have denormalized operands. In this case, all instructions that have issued out of the VSU Issue Queue and are younger than the denormalized instruction are invalidated and then reexecuted. To facilitate this, there is a Denormalized Instruction Queue that tracks all instructions that are issued from the VSU Issue Queue. When the internal flush condition is signaled, the VSU Issue Queue is stalled. Instructions in the Denormalized Instruction Queue are reissued one at a time (each instruction must complete before the next is issued).

6.3.3. Dual-Issue

This processor supports dual issue of instructions (two instructions can be issued in a cycle). Instructions can dual issue from either issue/stall point (IS2 and VQ8). Certain restrictions exist on what instruction combinations can be dual issued. See *Section 10.5.1 Instruction Issue* on page 99 for further details.

6.3.4. VSU Issue Queue

Figure 6-5 on page 30 is a block diagram of the VSU issue queue (VIQ). The VIQ collectively refers to the queues depicted on the left side of the diagram along with the VXU and FPU load miss queue (VMQ). The diagram shows the LSU pipeline on the right side in order to show how VXU and FPU loads are handled relative to the VIQ.

Instructions are sent in pairs to the VIQ in cycle IS2. If there are no VSU instructions to issue, then a "bubble"propagates down the queue. The queue does not "compress" out the bubbles. In other words, the queue is really a pipeline.
The VL1 and VL2 stages of the queue represent an overflow buffer. When a VQ8 stall occurs, this buffer continues to allow instructions to flow for an additional two cycles.
The IS2 point stalls two cycles after a VQ8 stall if and only if there is a VSU instruction that would overflow the VIQ if issued. Otherwise, the IS2 point can continue issuing FXU, LSU, and BRU instructions.

After VQ8, instructions are sent to the VXU or FPU. The VQ9, VQ10, VQ11, and VQ12 stages are transmit cycles that are required to physically send the data to the execution unit. See *Section 10.5.1 Instruction Issue* on page 99 for details of the dual-issue restrictions at the VQ8 stage.

FPU and VXU instructions can have denormalized operands. Therefore, a denormalized recycle queue is maintained to track each instruction past issue until the VXU or FPU signals that no internal flushes are necessary for the instruction. Instructions are issued to the execution units and to the denormalized recycle queue in parallel.
If no such denormalized condition occurs, then the instruction completes and is dropped from the denormalized recycle queue.
If an internal flush occurs, then the instruction and all younger instructions that are past VQ8 are reissued to the VSU in a single-step fashion until the denormalized recycle queue is empty.

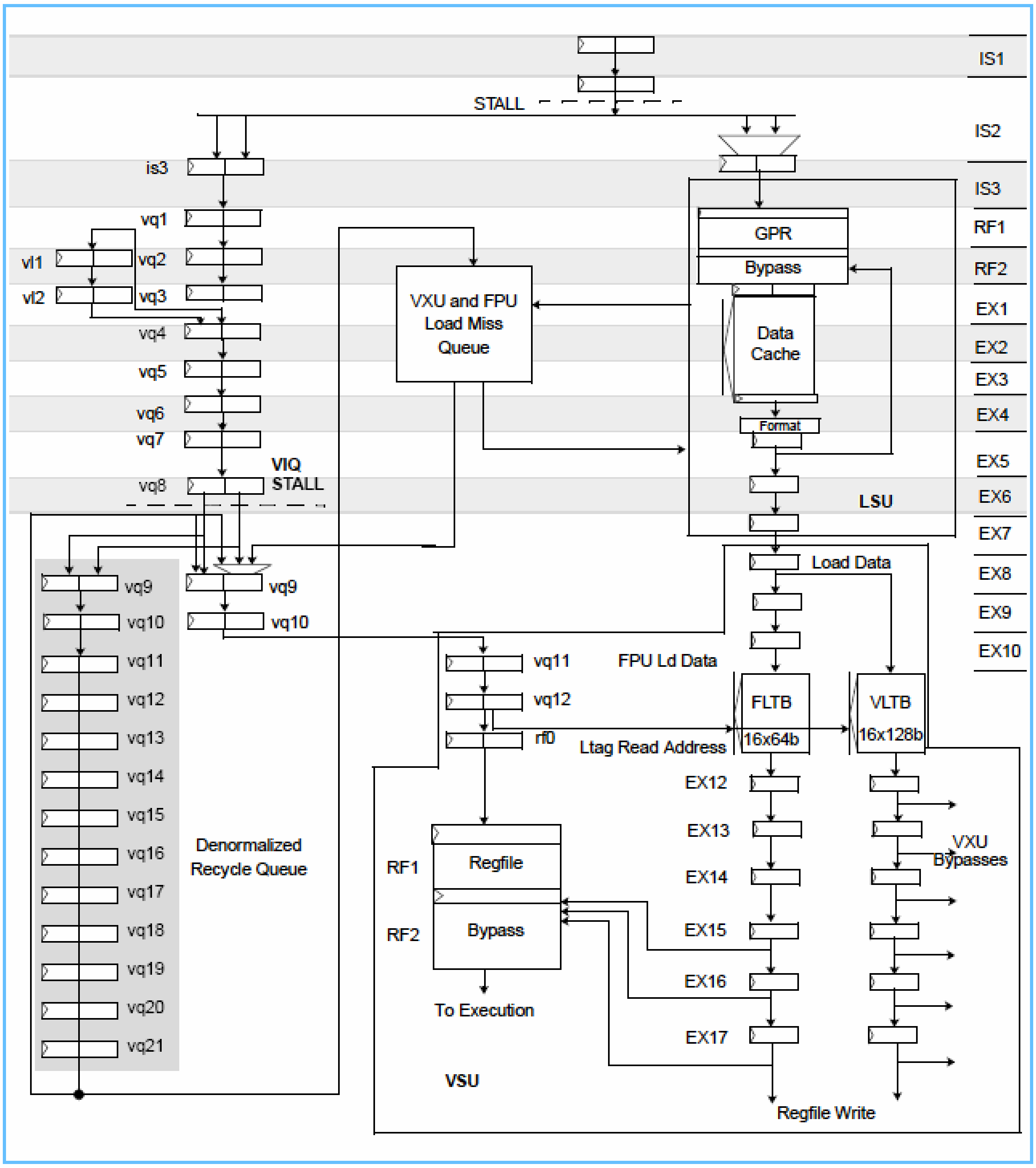
VXU and FPU loads are sent to the LSU to access the L1 data cache, in addition to being sent to the VIQ.
Once data is available for these loads, it is sent to the VXU load target buffer (VLTB) or FPU load target buffer (FLTB) as appropriate for the type of instruction. The load data is then held in these buffers until the instruction is issued from the VIQ. The VLTB and FLTB are 16 entries deep to accommodate a total of 16 loads in flight past the IS2 issue point.

If a VXU or FPU load misses the L1 data cache, it is issued to the VMQ. It is held in the VMQ until the cache miss is resolved in the LSU. Once the miss is resolved, the load is reissued to the VSU.

Internal flushes are caused in the VSU for the following reasons:

- An FPU or VXU floating-point instruction with a denormalized operand is encountered.
- An FPU instruction with a Not a Number (NaN) operand (as defined in *Book I: PowerPC User Instruction Set Architecture*) is encountered.
- An FPU instruction uses a bypassed suppressed result caused by an FPU enabled exception.
- The zero divide or invalid operation exceptions are enabled in the FPU when both were previously disabled.

Figure 6-5. VSU Issue Queue



6.3.5. Instruction Fetch

The Instruction Unit (IU) fetches a group of up to four aligned instructions from the 32 KB, 2-way set-associative L1 instruction cache.
Each thread has five instruction buffers in a FIFO queue configuration, and each instruction buffer (IBuf) holds a single fetch group, which can contain up to four instructions.
From the instruction buffers, two instructions are dispatched in each cycle to the instruction decode and dependency logic. They are then passed on to the issue pipeline stages.
Fetches for a particular thread only occur on alternating cycles. Therefore, the same thread never fetches on consecutive cycles, even in single-threaded mode.

When an IBuf becomes free (either when it has dispatched all of its instructions or after a flush), a fetch can be initiated for that thread. Either seven or eight cycles are required from the time the fetch is initiated until the instructions enter the IBuf, depending on the phase of the even or odd cycle with respect to the flush.

Fetch groups coming out of the L1 instruction cache are sent to the instruction buffers and also to the branch prediction logic.
Immediately following a pipeline flush, the instruction buffers for the flushed thread are empty, and the IU can fetch multiple fetch groups. In order to minimize latency, the IU speculatively fetches the next consecutive fetch groups in an attempt to fill the instruction buffers for the thread. However, before instructions reach the branch prediction logic, it is not known whether branches are present in a fetch group.
If it is determined that one of these groups contains a branch that is predicted taken, all instructions following the first predicted taken branch are invalidated and fetching is redirected to the predicted target address.
The latency in the IU for predicting a branch as taken and fetching the new instructions is eight cycles (assuming L1 instruction cache and instruction effective-to-real-address translation [I-ERAT] hits).

Programming Note:

Fetch groups are always quadword aligned, and the PPE does not merge instructions from partially empty instruction buffers into a single IBuf. For this reason, fetching is most efficient when branch targets point to the first instruction in a 4-instruction fetch group, and taken branches are the last instruction.

On instruction fetches, the least significant bits of the effective address (EA) are used directly to simultaneously index into the L1 instruction cache, the L1 instruction cache directory, the effective-to-real-address translation table (I-ERAT), and the branch history table (BHT).
Several conditions need to be satisfied to consider the fetch a hit:

- The EA of the instruction must match the EA contained in the I-ERAT entry being indexed. The I-ERAT entry must also be valid.
- The Instruction Relocate (IR) and Hypervisor State (HV) bits from the Machine State Register (MSR) must match the corresponding bits in the I-ERAT, which were set at the time the I-ERAT entry was loaded.
- The thread performing the fetch must match the thread ID from the I-ERAT entry.
- The Real Address (RA) from the L1 instruction cache directory must match the RA provided by the I-ERAT.
- The Page Protection bits allow this access to occur.

If all of these conditions are met, and if no L1 instruction cache parity errors have occurred, the fetch is a hit.

6.3.5.1. Instruction Cache Misses

When an L1 instruction cache miss occurs, a request is made to the L2 for the cache line. This is called a “demand fetch.” A demand fetch can proceed to the L2 before the instruction causing the miss is committed (“committed” means that all older instructions are past the flush point). However, to proceed beyond the L2, the instruction must be committed.

If the request hits in the L2, the 128-byte line is reloaded approximately 36 cycles after the request, as long as there is no contention in the L2. For more detail, see *Section 7.4 L2 Cache* on page 60.
The reloaded data is returned on four consecutive clock cycles and buffered in a cache line buffer (CLB). Half of the cache line (64 bytes) is written into the L1 instruction cache at a time. Because it is single ported and a thread can only access the L1 instruction cache on alternating cycles, three cycles are required to do the write (one cycle to write the first half, one cycle pause for the other thread, one cycle to write the second half).

To improve performance, the first beat of data returned by the L2 contains the fetch group with the address that was requested. Therefore, it returns the data critical sector first. To reduce the latency of an L1 instruction cache miss, this critical sector of the cache line is sent directly to the IC4 stage of the pipeline, instead of waiting to write it into the L1 instruction cache and then reread it (this is termed bypassing the cache).

Each thread can have up to one instruction demand fetch and one instruction prefetch (see *Section 6.3.5.2*) outstanding to the L2 at a time. This means that, in multithread mode, up to four total instruction requests can be pending simultaneously.

Note: In multithreading mode, an L1 instruction cache miss for one thread does not disturb the other thread.

6.3.5.2. Instruction Prefetch

In order to prevent performance loss due to L1 instruction cache misses, the PPE uses speculative instruction fetch for the L1 instruction cache and instruction prefetching to the L2 cache. Because the PPE can fetch up to four instructions per cycle and can execute up to two instructions per cycle, the IU often has several instructions queued in the instruction buffers. Letting the IU speculatively fetch ahead of execution means that L1 instruction cache misses can be detected early and fetched while the processor remains busy with instructions in the instruction buffers.

In the event of a cache miss, a request for the required line is sent to the L2. In addition, the L1 instruction cache is also checked to see if it contains the next sequential cache line. If it does not, a prefetch request is made to the L2 cache to bring this next line into the L2 (but not into the L1, to avoid L1 instruction cache pollution). This prefetch occurs only if the original cache miss is “committed” (in other words, it must be certain that all older instructions have passed the flush point).
This is especially beneficial when a program jumps to a new or infrequently used section of code, or following a task switch, because prefetching the next sequential line into the L2 hides a portion of the main memory access latency.
Prefetch requests are not made when a page might potentially be crossed, meaning that the next sequential cache line is on a different aligned 4 KB boundary.

Note: Prefetching is not performed on instructions in “caching-inhibited” space (I = ‘1’).

6.3.6. Branch Prediction

In order to reduce the penalty of branch misprediction, the IU attempts to predict both the direction (taken/not taken) and target address for branches using several mechanisms.

Branch direction can be predicted either dynamically using the branch history table (BHT) or statically with the "a" and "t" bits provided by the PowerPC Architecture. This approach is described in the branch instructions section of *Book I: PowerPC User Instruction Set Architecture*.
Conditional branches that have a Branch Options (BO) field of '1z1zz'are considered unconditional, because the branch is always taken and the direction is not considered to have been predicted.

Branches that use the immediate field of the branch instruction to indicate the target address never mispredict the target address, because it can be computed in the branch predict pipeline.
Branch-to-link instructions do require the target address to be predicted using the link stack; branch-to-count instructions do require the target address to be predicted using a shadowed copy of the Count Register (CTR). The Link Register (LR) and CTR prediction mechanisms are described in *Section 6.3.6.3* on page 34 and *Section 6.3.6.4* on page 34.

Because of the high-frequency nature of the PPE and the corresponding long pipeline, the branch misprediction penalty can be significant. When a branch reaches the end of the Branch Unit (BRU) pipeline and is discovered to have been mispredicted, several cycles are required before that thread begins fetching along the correct address. A branch misprediction can be detected on any cycle, but a thread fetches on alternating cycles. Therefore, the total mispredict-penalty can be either 23 or 24 cycles depending on whether the flush occurred on an even or odd cycle.

Special logic is used to detect conditional branches to the next sequential instruction; that is, **bc .+4**. Regardless of whether these branches are predicted correctly, they never cause a flush or update the BHT.

6.3.6.1. Branch History Table

The branch history table (BHT) is organized as 4 K-entry array (each entry has 2 bits). It is accessed in parallel with the L1 instruction cache. Using 2 bits for each BHT entry allows the processor to store one of the four states: strongly not taken, weakly not taken, weakly taken, strongly taken.
The Global Branch History (6 bits) is constantly updated with the latest speculative predicted history value. In the event of a pipeline flush including branch misprediction, the correct history state is restored.
The index address into the BHT is generated by XORing bits [50:61] of the fetched instruction address with the Global Branch History. In other words, `(EA[50:55] XOR Global_Hisory[0:5] || EA[56:61]).`
Only instructions that relied on the BHT to predict their direction ever cause an update to the BHT. This means that unconditional branches (including conditional branches with BO = '1z1zz') and statically predicted conditional branches do not update the BHT.

6.3.6.2. Branch Prediction Using Static Prediction and "a" and "t" Bits

Some conditional branches are known ahead of time to be almost always unidirectional.
Software can communicate this to the processor by setting the "a" bit in the BO field of the branch instruction.
If the "a" bit is '0', then the branch is predicted dynamically using the BHT as described above.
If the "a" bit is '1' in the branch, the BHT is not used, and the BHT is not updated if this branch is mispredicted.
The direction of the static prediction is in the "t" bit of the BO field, which is set to '1' for taken and to '0' for not taken.
Software is expected to use this feature for conditional branches where it believes that static prediction will be at least as good as hardware branch prediction.

Programming Note:

Static prediction can reduce pollution in the BHT, improving accuracy on branches elsewhere in the code.

6.3.6.3. Address Prediction Using the Link Stack

The link stack is used to predict the target address of branch-to-link instructions. The PPE has two independent 4-entry link stacks, one for each thread.

When the branch prediction logic detects a branch and link instruction that is either unconditional or conditional and predicted taken (**bl**, **bcl**, **bcla**, **bclrl**, or **bcctrl**), the address of the next sequential instruction is pushed onto the link stack.
When a subsequent branch-to-link instruction, such as bclr or bclrl, is fetched and predicted taken by the branch prediction logic, and also has hint bits that indicate a subroutine return (BH = '00'), the address at the top of the link stack is used to predict the target address.
This address is then popped from the top of the link stack and discarded.

Programming Note:

- When several consecutive branch and link instructions are executed without intervening branch-to-link instructions, the link stack may overflow and wrap around, overwriting the oldest addresses.
- Conditional branch-to-link instructions that have the next instruction as the target (that is, **bcl .+4**) do not update the link stack, regardless of whether the branch was taken. This can be used as a programming aid to store the current instruction address in the Link Register.

Pipeline flushes are a potential problem for the link stack, because when the head of the link stack is set incorrectly, several mispredicts of branch-to-link instructions can occur. For this reason, the link stack logic contains mechanisms to recover the value of the link stack pointer in the event of a flush.

6.3.6.4. Branch-to-Count Instructions

The target addresses of branch-to-count instructions are predicted with a shadowed copy of the CTR. The value in the CTR is updated by either the execution of an **mtspr CTR** instruction or by a branch that decrements the CTR (described in the Branch Processor Section of *Book I: PowerPC User Instruction Set Architecture*).
Then, the new value is reflected in the shadowed copy when the CTR-updating instruction reaches the EX4 stage of the Branch Unit pipeline.

Note:

The shadowed copy of the CTR is used to predict the target address of a branch-to-count instruction, such as **bcctr** or **bcctrl**. However, it is not used to predict whether a branch that relies on the CTR, such as **bdnz**, is taken or not taken. This is determined either with static prediction as described in Section 6.3.6.2 or by using the BHT as described in Section 6.3.6.1.

Programming Note:

For the target address of a bcctr instruction to be predicted correctly, the **mtspr CTR** should be placed as far ahead in the instruction stream as possible to give the shadow copy of the CTR time to be updated.

6.4. Fixed Point and Load/Store Execution Unit (XU)

The execution unit (XU) contains the FXU, LSU, and MMU. This section gives a high-level overview of these units. The FXU is responsible for all integer operations except for load or store types of instructions. The LSU handles all data accesses (load and store instructions) including the L1 Data Cache and primary address translation via the D-ERAT. Finally, the MMU implements secondary address translation.

Figure 6-6 on page 37 shows a dataflow diagram of the LSU and FXU. The front end of the pipeline consists of a shared register file, the GPR (there are 2 read ports for the LSU, and 5 read ports for the FXU). All dependency checking is taken care of in the IU before instructions are issued; thus, once the instruction reaches the XU, there are no stall points. Any execution problems that arise in the XU results in a flush that is signaled at pipeline stage EX7. If no flush occurs, results of the instruction execution are written to the GPR in cycle EX9.

Both the FXU and LSU require two GPR read cycles, RF1 and RF2, where GPR data is accessed and distributed. RF2 is the bypass cycle where data for the instruction is either selected from the GPR or “bypassed” from an older instruction that has results that have not yet been written back to the GPR.

6.4.1. Load Store Unit (LSU)

The LSU pipeline starts with a register file access (RF1), followed by a forwarding stage (RF2), and then execution. The LSU pipeline accesses the DCache and takes four pipeline stages. This starts with an address generation (AGEN) stage followed by three access and result forwarding stages (EX2 - EX4).

The execution stages, EX1 through EX5, comprise the L1 complex stages of the LSU. During these cycles, an effective-to-real-address translator (ERAT), a 4-way L1 DCache, and a Tag (or cache directory) are accessed. ERAT, tag, and cache accesses start in parallel in EX1, and the end result is available in EX5. The DCache tag and ERAT are accessed in parallel to check for a hit or miss. Any load or store that misses the ERAT causes all subsequent instructions to flush while the ERAT attempts to reload the translation from the MMU. While the MMU is being accessed, the pipeline refills up to the issue stall point. The bypass stages, EX6 through EX10, allow the resultant load data to be bypassed around the GPR. The GPR is then updated with the load data in EX9 and EX10.

For load instructions, if the instruction hits in the tag or ERAT, then the LSU returns the data from the DCache. It writes the data to the GPR Register File. If the load has an L1 DCache miss, its address and control information are stored in the load miss queue until data returns from the memory subsystem. The miss queue wins arbitration to recycle this load back through the pipeline so that the GPR can be updated properly as in the case of an L1 DCache hit. The miss queue is eight entries deep. All eight entries can simultaneously access the L2 cache.

Stores are issued to both the LSU and FXU pipeline. The store address uses the LSU EX1 through EX5 stages to translate the effective address and look up the tag. The store data runs down the FXU pipeline and is sent to the Store Queue. Store instructions flow through the same pipeline as loads and access the tag or ERAT in parallel. Data for the store instruction is read from the GPR Register File and written to the store queue (STQ). After the store address and data are written to the STQ and the tag or ERAT indicates a hit, the store can write to the DCache. If the store misses the tag, it is moved to the LMQ.

The store real address and store data are queued until all older instructions in the XU have completed. Stores attempt to access the DCache when they are the oldest instruction and the cache is either idle (no higher-priority load) or another store instruction is accessing the tag or ERAT. A store can, however, go ahead of an older load if it is to a different congruence class (bits 52:56 of the address are different). This feature applies only to cacheable stores that are not synchronization, coherency, or cache management operations. The L1 DCache is a “write through” cache. Therefore, when the store is the oldest and then wins arbitration for the L1 DCache, the data is sent to the L2 cache. It updates the L1 DCache if necessary. The L1 DCache is only written when a hit occurs. The store queue is 16 entries deep.

6.4.2. Fixed-Point Integer Unit (FXU)

The primary subunits in the fixed-point integer unit (FXU) are listed below and appear in *Figure 6-6 XU Dataflow Diagram* on page 37 and *Figure 6-7 FXU Block Diagram* on page 38:

- General Purpose Register (GPR) — This is a 5-read and 2-write register file with 32 entries. One write port is dedicated to the FXU instructions, and the other write port is dedicated to the LSU. The RA and RB read ports are separate for the FXU and the LSU, and the RS read port is for store and **mfspr** instructions. The GPR is logically duplicated per thread.
- Arithmetic Logical Unit (ALU) — Supports add, subtract, compare, and logical operations.
- Count Leading Zero Unit (CLZ) — Supports count leading zero instructions.
- Rotate — Supports shift, rotate, sign extend, decimal, and select operations.
- Multiply — Supports all forms of multiply.
- Divide — Supports all forms of divide.

The FXU is delayed three cycles beyond the LSU pipeline. This delay occurs after the FXU instruction is issued and before the GPR is accessed. This delay covers up most of the load-use latency between a load and a fixed-point operation dependent on the load, which is a common case in critical code. After the three-cycle delay, three operands are read from the GPR in EX2 and EX3 (which are similar to the LSU's RF1 and RF2). EX3 is the bypass stage where the proper data for the instruction is selected from the GPR or from a target bypass stage of an older instruction. Most FXU instructions are 2-cycle pipelined operations, meaning the computation of the operation occurs in EX5 and bypassing of target register results can occur as early as cycle EX6. The writeback of the fixed-point instructions is delayed until after the final LSU exception point. Hence, EX6 through EX10 are bypass stages (in order to match the LSU). At EX9 the result data is written to the GPR.

In addition to *Figure 6-6*, a more detailed close-up view of the FXU can be seen in *Figure 6-7 FXU Block Diagram* on page 38. This more detailed diagram shows the multiplexing that occurs to implement the bypass feature of the pipeline.

Certain "complex" instructions, which include multiply, divide, and move to and from Special Purpose Registers (SPRs) instructions, take many cycles to execute and are not pipelined. When one of these instructions is issued, the IU stalls until it is completed. For performance details see *Section 10 Performance* on page 88.

Figure 6-6. XU Dataflow Diagram

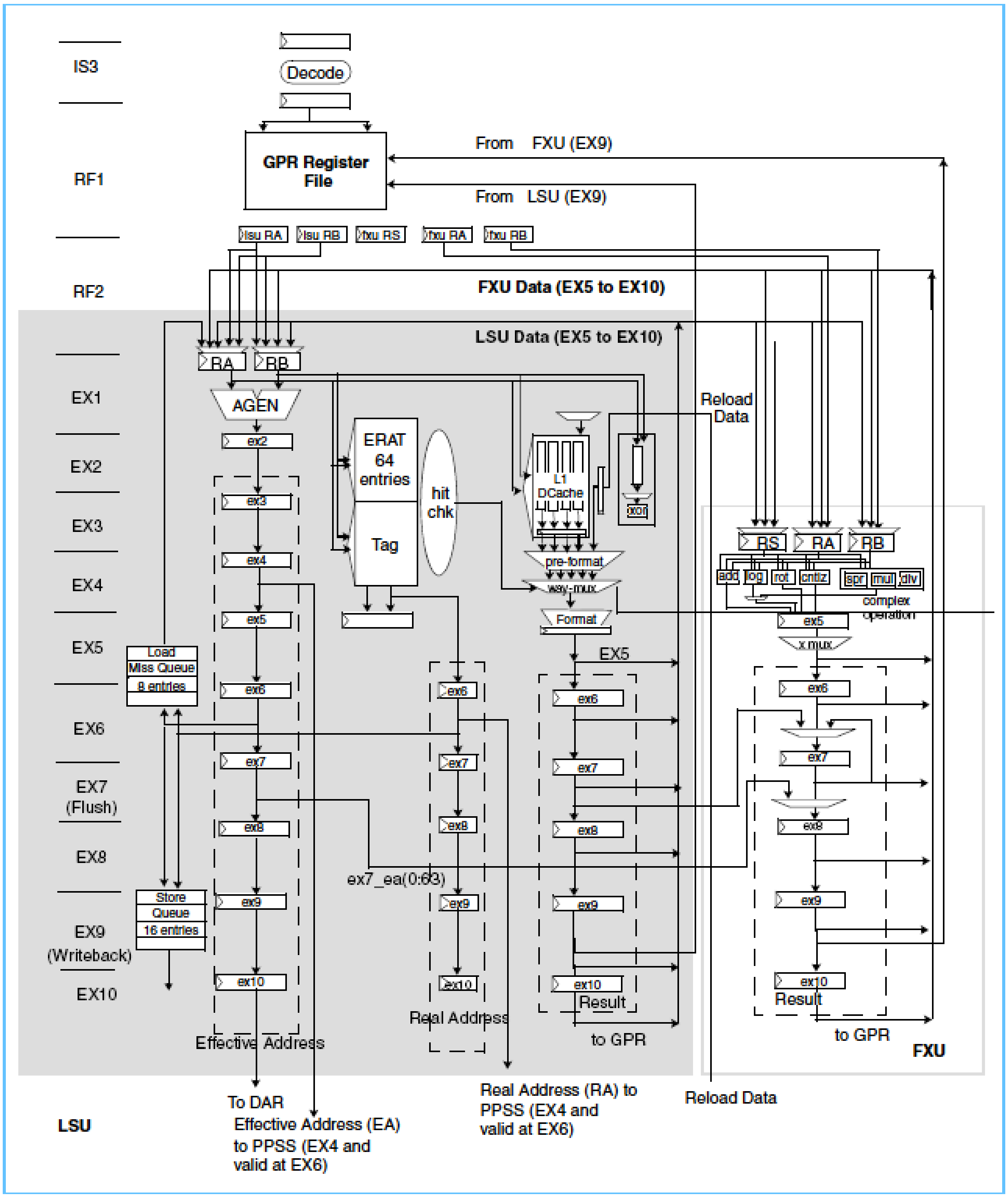
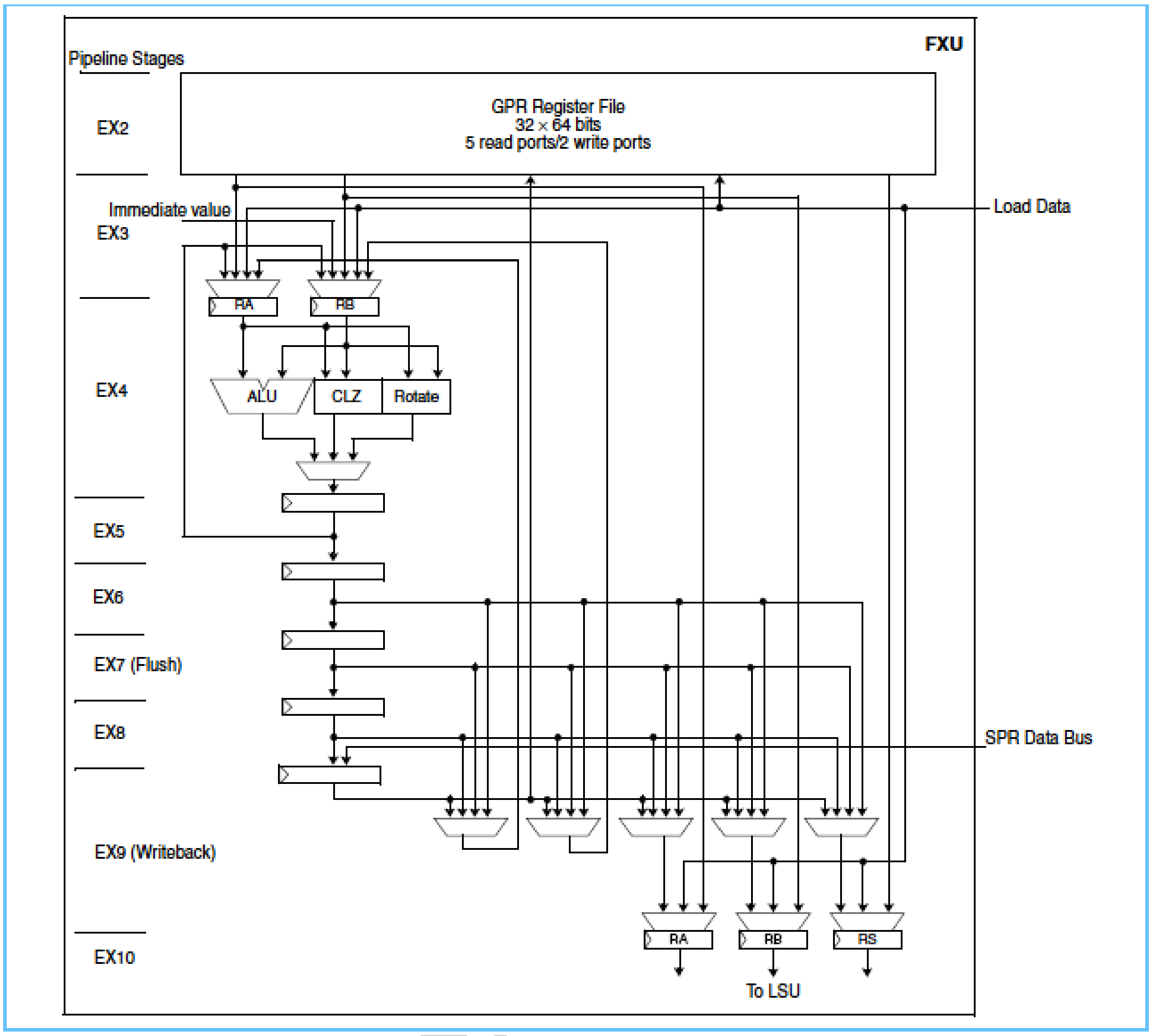


Figure 6-7. FXU Block Diagram



6.4.3. Memory Management Unit

The Memory Management Unit (MMU) provides address translation for the PPU. The MMU is a second level address translation facility. The PPE provides the first level of translation by separate instruction effective-to-real-address translation (I-ERAT) and data effective-to-real-address translation (D-ERAT) arrays that are smaller and faster than the MMU.

When an address is not found in either ERAT, it must be sent to the MMU to resolve. The MMU first determines if the address is in real mode or virtual mode based on the Instruction Relocate and Data Relocate bits in the Machine State Register (MSR[IR] or MSR[DR] respectively) as appropriate. If in real mode, the MMU translates the address directly. If in virtual mode, the address is translated using the SLB and TLB. Details can be found in *Section 8 Memory Management* on page 67.

6.4.3.1. Features

The following list summarizes the implemented functions of the MMU.

- Virtual Address Translation
  - 64-bit effective address (EA)
  - 65-bit virtual address (VA)
  - 42-bit real address (RA)
- Segment Lookaside Buffer (SLB) for effective-to-virtual-address translation
  - Two logical SLB arrays, one per thread
  - Each array has 64 fully-associative entries
  - Software management via **slbmte**, **slbmfee**, **slbmfev**, **slbie**, and **slbia** instructions

  For more information, see *Section 8.3 Segment Lookaside Buffer (SLB)* on page 69.
- Translation Lookaside Buffer (TLB) for virtual-to-real-address translation
  - 1,024 total entries
  - Four-way set-associative
  - Pseudo (binary tree) least recently used (LRU) default replacement policy (256 entries with three bits for each entry)
  - Replacement management table allows software to change the replacement policy (eight classes with four bits for each class)
  - Hardware TLB-miss resolution (can be enabled and disabled using the TLB load bit of the Logical Partition Control Register, LPCR[TL])
  - Software management via **mtspr**, **mfspr**, **tlbie**, and **tlbiel** instructions
  - Entries are tagged with logical partition IDs for sharing of the TLB among multiple partitions

  For more information, see *Section 8.4 Translation Lookaside Buffer (TLB)* on page 70.
- Large Page Support
  - Three concurrent system page sizes (4KB and two of the following: 64 KB, 1 MB, or 16 MB)

  For more information, see *Section 8.4.2 Large Page-Size Decode* on page 74.
- Real Address Translation
  - Real mode storage control support
  - Hypervisor mode support (via Hypervisor Real Mode Offset Register [HRMOR])
  - Privileged mode support (via Real Mode Offset Register [RMOR] and Real Mode Limit Selector [RMLS])

The MMU is a serialized execution unit in that it handles one translation request at a time. The processor sends up to one I-ERAT and one D-ERAT miss to the MMU for translation. If two requests are pending, the MMU saves the second request until the first request is completed. If a TLB miss occurs for the first request (and the TLB is in hardware tablewalk mode), then the second request is allowed to proceed while the first is pending resolution. If the second request misses the TLB as well, then the MMU ignores the miss and retries the translation again later once the first miss has been resolved.

The MMU requires 9 cycles to perform a translation, as shown below in *Figure 6-8*. There are additional two cycles between the I-ERAT or D-ERAT and the MMU, so the total time for a reload is 13 cycles. The diagram shows the dataflow path for a virtual address translation request as well as a short description of what function is performed at each stage. A real address translation request takes the same amount of time, but does not involve the SLB or TLB, which are bypassed.

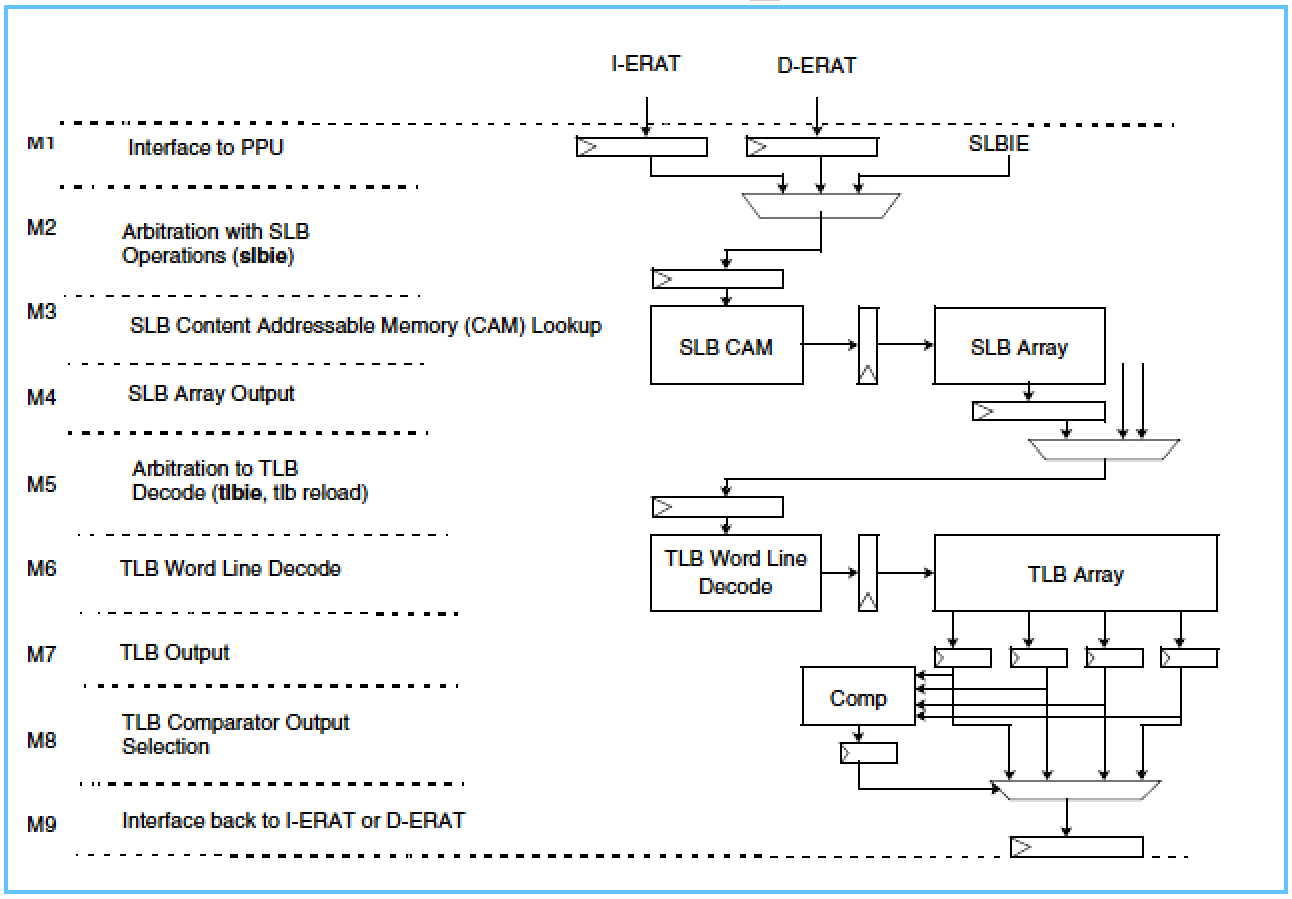
6.4.3.2. Software Management

The MMU also handles software access to the SLB and TLB. The SLB is accessed through the PowerPC Architecture instructions: **slbmte**, **slbmfee**, **slbmfev**, **slbie**, and **slbia**. These instructions are sent to microcode engine, which converts them to sequences of move-to and move-from Special Purpose Register (SPR) instructions. The MMU receives these commands and performs the appropriate function on the SLB (for the appropriate thread).

The TLB is accessed by software by issuing direct move-to and move-from SPR instructions. This facility is documented in *Section 12.3 TLB Management Instructions and ERAT Coherency* on page 125.

Since instructions that access the SLB or TLB directly are considered to be rare in performance-critical code, these instructions can take approximately 30 cycles to complete. If the MMU is busy handling a translation request, the MMU queues the software-initiated SPR access until the translation request is complete. If a translation request misses in the TLB, the MMU allows the SPR access to occur while the TLB reload is pending.

Figure 6-8. MMU Dataflow



I-ERAT
D-ERAT
M1
Interface to PPU
SLBIE
M2
Arbitration with SLB Operations (**slbie**)
M3
SLB Content Addressable Memory (CAM) Lookup
SLB CAM
SLB Array
M4
SLB Array Output
M5
Arbitration to TLB Decode (**tlbie**, tlb reload)
M6
TLB Word Line Decode
TLB Word Line Decode
TLB Array
M7
TLB Output
Comp
M8
TLB Comparator Output Selection
M9
Interface back to I-ERAT or D-ERAT

6.5. Vector/Scalar Unit (VSU)

The VXU and FPU are collectively referred to as the VSU (Vector-Scalar Unit). *Figure 6-9 VSU Dataflow Diagram* on page 42 shows a high level dataflow diagram of the VSU.

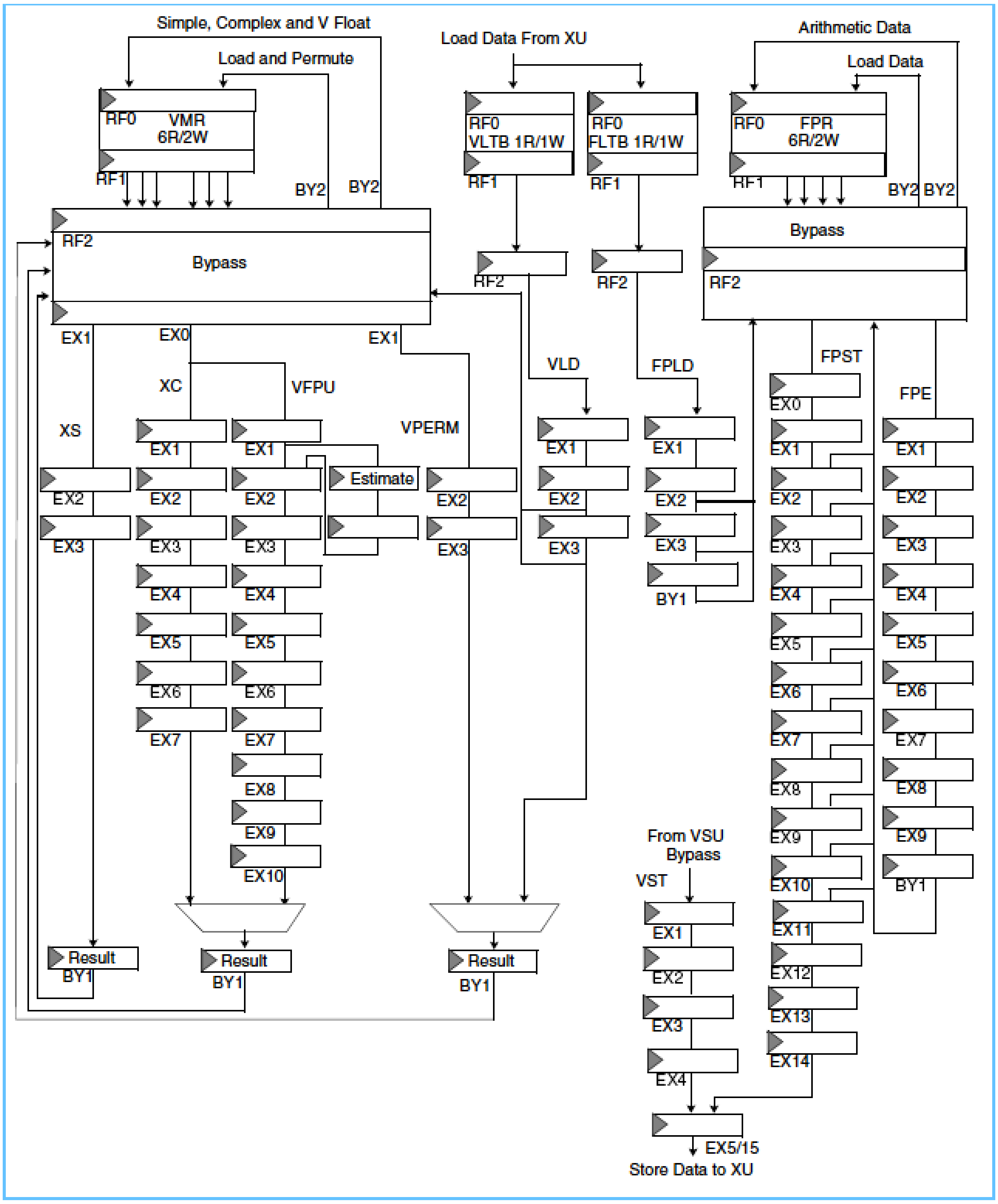
Six execution pipelines make up the VXU: Simple (XS), Complex (XC), Vector Floating-Point Unit (VFPU), Vector Permute (VPERM), Vector Load (VLD) and Vector Store (VST). A listing of which instructions are issued to each pipeline is found in *Table 10-2 VXU Instruction Characteristics* on page 94. The Vector Multimedia Register (VMR) holds the architected operand and target registers. The Vector Bypass provides operands to each pipeline (either from the VMR or bypassed from results of the execution pipes) and directs data for write-back of results to target registers. Lastly, the Vector Load Transfer Buffer (VLTB) queues up returned load data for use in the Vector Load pipeline.

The FPU has three main execution pipes: Scalar Floating-Point Execution (FPE), Scalar Floating-Point Load (FPLD) and Scalar Floating-Point Store (FPST). Like the VMR, the Scalar Floating-Point Register (FPR) holds the architected operand and target registers. Scalar Bypass provides operands to each pipeline and directs data for write-back of results to target registers. Like the VLTB, the Scalar Floating-Point Load Transfer Buffer (FLTB) queues up returned load data for use in the Scalar Floating-Point Load pipeline.

Figure 6-3 on page 25 shows a pipeline diagram of the VSU beginning with the VIQ.

The VSU has two issue slots, which allow for simultaneously issuing two instructions. The first slot (i0) can issue VFPU, XS, XC, and scalar FPU instructions. The second slot (i1) can issue permute instructions and vector and scalar loads and stores. The VMR provides six 128-bit busses of data to the pipes: Operands A, B, and C for issue slot i0 and again for slot i1. This requires VXU Bypass to also provide 6 output busses. The Bypass outputs can be sourced from the VMR, the VLTB, or from the result of any of the pipelines described above.

Figure 6-9. VSU Dataflow Diagram



6.6. PowerPC Processor Unit Multithreading Implementation

The PowerPC Processor Unit (PPU) allows for two simultaneous threads of execution within the processor and can be viewed as a two-way multiprocessor with shared dataflow. This gives the effective appearance of two independent processing units to software.

The performance of the two threads is limited, however, because they must share resources such as the L1 and L2 caches. Additionally, both threads execute in the same logical partition.

This section describes the multi-threaded capabilities of the PPU.

6.6.1. Resources Shared or Duplicated between Threads

The PPU duplicates all architected state, which includes all architected registers and Special Purpose Registers (SPRs) with the exception of registers that deal with system-level resources, such as logical partitions, memory, and thread-control. This creates the illusion to software that two separate virtual processors are available for instruction execution. Resources that are not described in the PowerPC Architecture are usually shared for both threads, except in cases where the resource is small or offers a critical performance improvement to multithreaded applications. The following sections indicate which resources are shared or duplicated.

Programming Note:

Both processor threads share logical partitioning resources and therefore must always execute in the same logical partition context. For a list of logical partitioning resources, see *Section 1.7* of *Book III: PowerPC Operating Environment Architecture, Version 2.02*.

6.6.1.1. Arrays, Queues, and Other Structures

Since arrays and queues are typically large chip-area logic components, they are usually shared between threads.

The following arrays, queues, and structures are fully shared between threads.

- L1 Instruction Cache (ICache) array
- L1 Data Cache (DCache) array
- Instruction Effective-to-Real-Address Translation (I-ERAT) array
- Data Effective-to-Real-Address Translation (D-ERAT) array
- Translation Lookaside Buffer (TLB) array
- L2 Cache array
- Load miss queue
- Store queue
- I-ERAT miss queue
- D-ERAT miss queue
- Microcode Engine
- Instruction Fetch Control
- PowerPC Processor Storage Subsystem (PPSS)
- All execution units (Branch [BRU], Fixed-Point Integer Unit [FXU], Load and Store Unit [LSU], Floating-Point Unit [FPU], Vector/SIMD Multimedia Extension Unit [VXU])

The following arrays and queues are duplicated for each thread.

- Segment Lookaside Buffer (SLB) array
- Branch history table (BHT) array

- Global branch history (GBH) array
- Instruction Buffer (IBuf) queue
- Link stack queue

Splitting the IBufs allows for each thread to dispatch independently from the other. Splitting the SLB array is convenient for the implementation because of the nature of the PowerPC Architecture instructions that access it, and also because it is a relatively small array.

The instruction fetch control is shared for both threads because fetch alternates between threads every cycle. Each thread maintains its own BHT and GBH to allow each thread to do its own branch prediction independent of the other.

6.6.1.2. Registers

The following architected registers are duplicated for multithreading (see *Figure 2* in *Section 1.6 of Book I: PowerPC User Instruction Set Architecture, Version 2.02*):

- General Purpose Register (GPR) (32 entries per thread)
- Floating-Point Register (FPR) (32 entries per thread)
- Vector Multimedia Register (VMR) (32 entries per thread)
- Bookmark (BKMK), Condition Register (CR), Count Register (CTR), Link Register (LR), Fixed-Point Exception Register (XER), Floating-Point Status and Control Register (FPSCR), Vector Status and Control Register (VSCR) (1 per thread)

Microcode engine has 5 special GPR registers dedicated for microcode instructions. These registers are not visible to software and are used exclusively by the microcode engine. Since the microcode engine is not duplicated per thread, these registers are not duplicated either.

Table 5-1 PowerPC Processor Element SPRs on page 14 lists all implemented SPRs with a column showing if each register is shared or duplicated per thread.

All Memory Mapped I/O (MMIO) registers in the PPU deal with aspects of the PPSS that are shared for both threads, so all MMIO registers are shared for both threads as well.

6.6.2. Pipeline Sharing

Figure 6-10 shows the instruction flow for multithreading. Instruction fetch maintains a separate Instruction Fetch Address Register per thread (IFAR 0 and IFAR 1). Fetching alternates every cycle between threads. The instruction fetch is pipelined and requires eight cycles to restart fetching for a predicted-taken branch. The four new instructions fetched from the cache are forwarded to the Instruction Buffer for the thread that is fetching the instructions.

Instructions are sent to the shared decode, dependency, and issue pipeline in an alternating pattern determined by the thread priority (see *Section 6.6.3.2* on page 50). Having separate Instruction Buffers allows one thread to continue making progress if the other thread is stalled at dispatch.

From dispatch on, each pipeline stage contains instructions from one thread only in a given cycle.

Figure 6-10. Multithread Instruction Flow

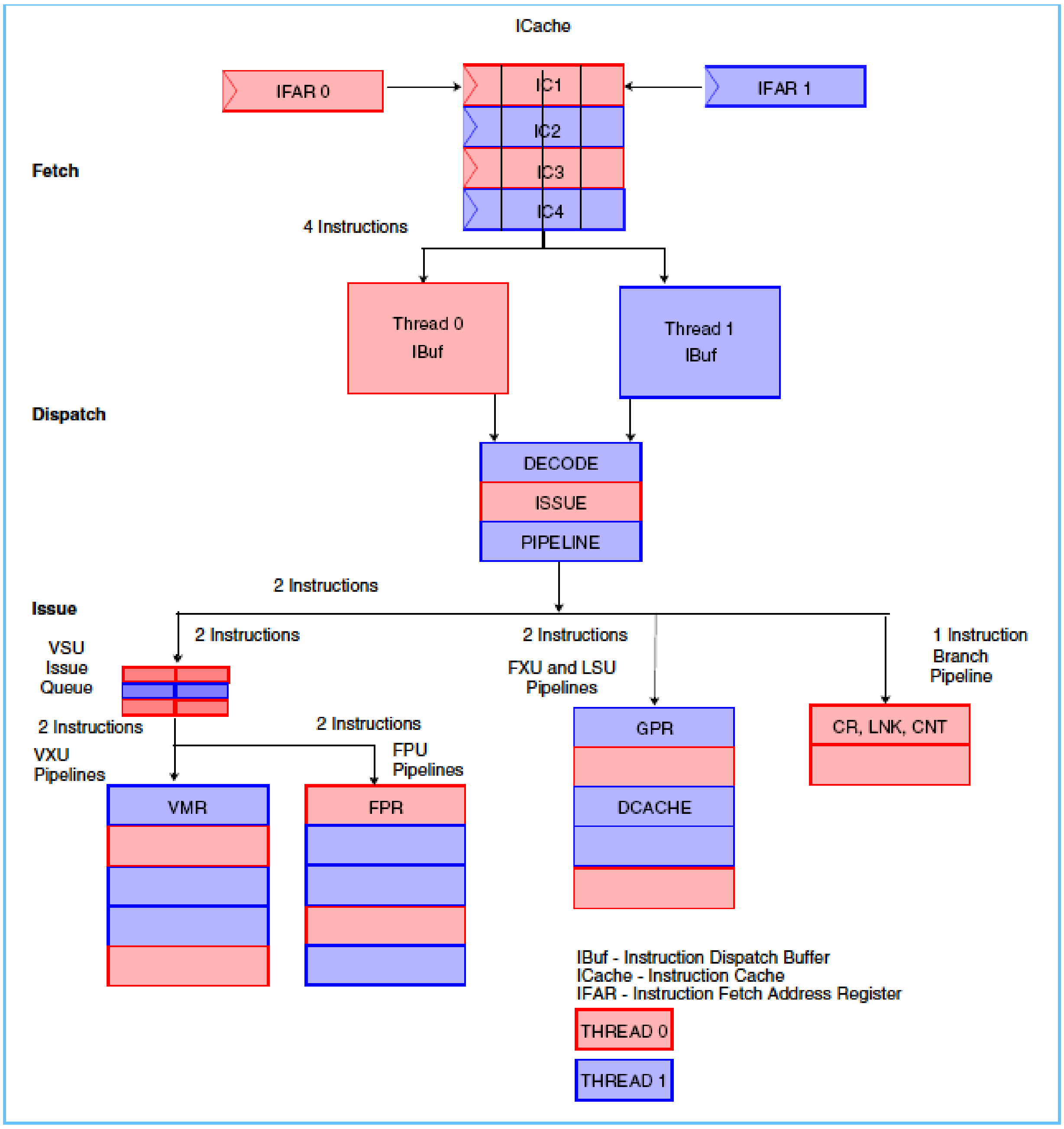


Figure 6-11 shows normal fetch, dispatch, and issue with no branch redirects or pipeline stalls. In this example, fetch, dispatch, and issue alternate between threads every cycle. T0[0:3] is the group of four instructions fetched for thread 0, and T1[0:3] is the group of four instructions fetched for thread 1.

Figure 6-11. Thread Switching with No Branches

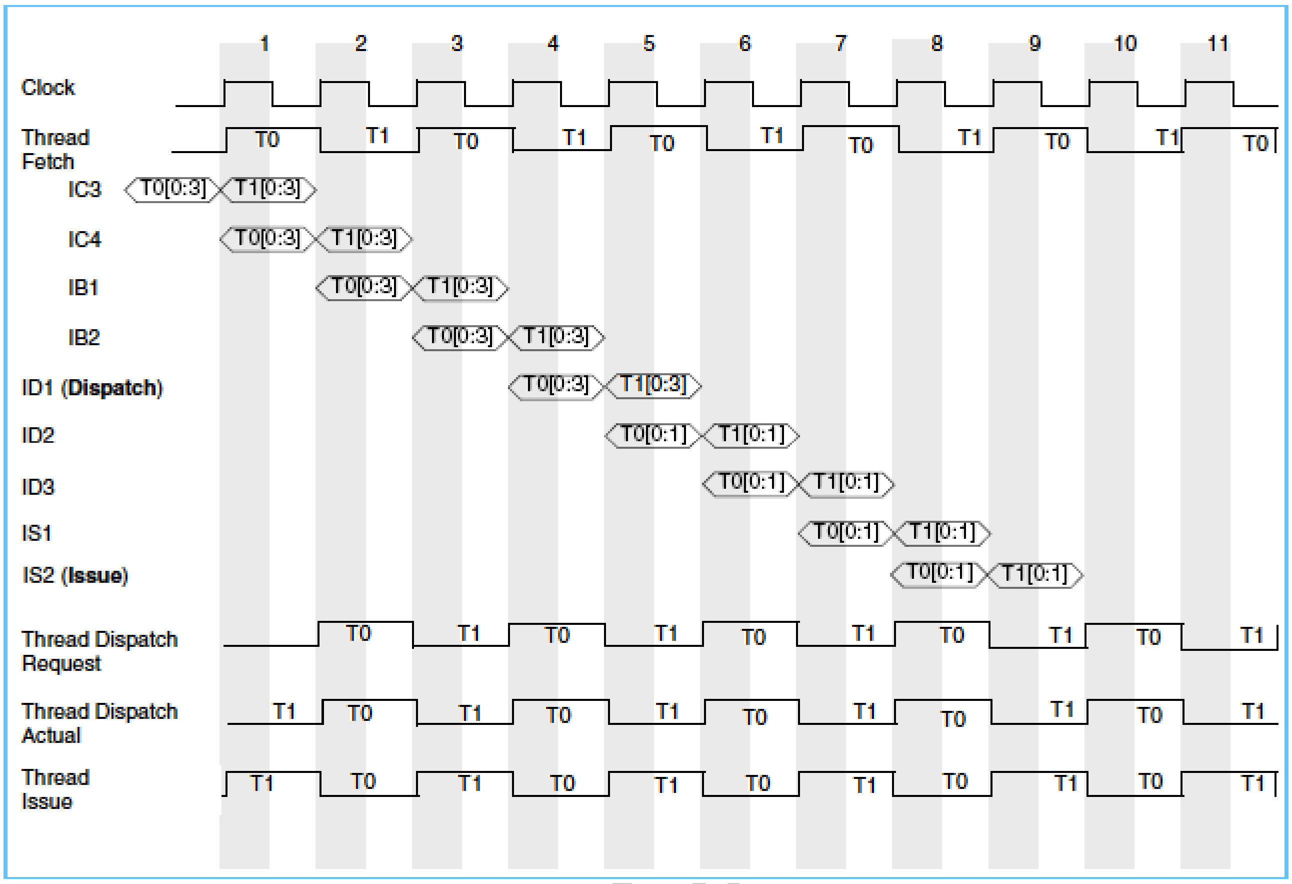
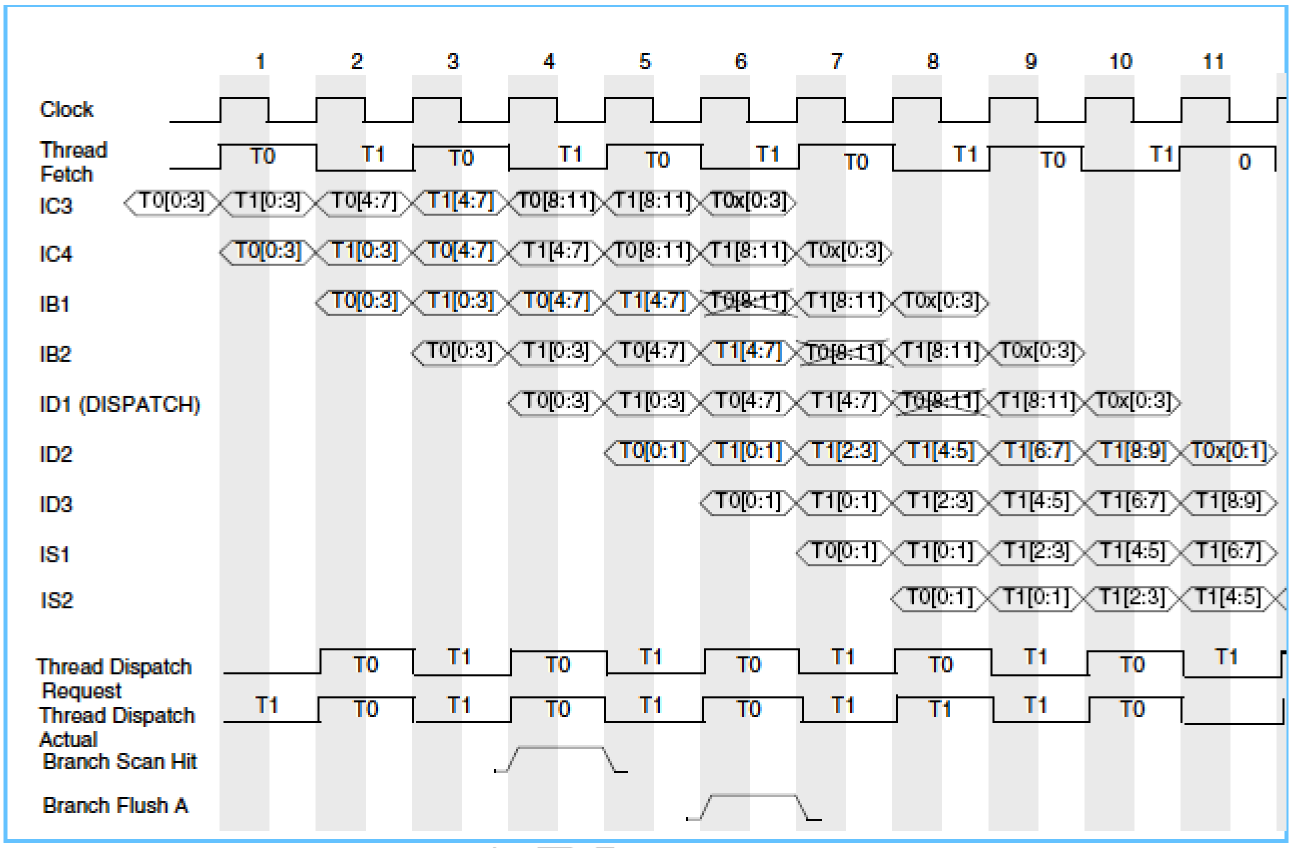


Figure 6-12 shows an example of a branch redirect. In this example, T0[1] is the second instruction in group T0[0:3] and is a branch which is predicted to the taken path. T0x[0:3] is the group of four instructions fetched from the branch target.

In cycle six, thread 0 is flushed due to the branch redirect. The instruction before the branch T0[0] and the branch T0[1] are issued. The instructions after the branch are flushed. This allows thread 1 to get three cycles of dispatch before thread 0 has instructions available on the target path.

Figure 6-12. Multithread with Branch on Second Instruction of Thread 0



6.6.3. Stalling, Flushing, and Prioritization

This section deals with multithreaded behavior in regards to stalling, flushing, and prioritizing instructions.

6.6.3.1. Stalling

There are three stall points in the pipeline: at dispatch (ID1), at issue (IS2), and at VSU issue (VQ8). The dispatch stall point is separate for each thread, allowing one thread to dispatch if the other thread is stalled. The IS2 and VQ8 stall points require both threads to stall. If a stall occurs in the VSU Issue Queue (at VQ8), then it stalls both threads if and only if it results in a stall at the IS2 issue point. If the IS2 point stalls, then dispatch is stalled for both threads regardless of what is in the pipeline stages between dispatch and issue.

An instruction dispatch from a thread can be stalled due to any of the following conditions:

- A dispatch stalling **nop** instruction is dispatched
- The other thread has higher priority (see *Section 6.6.3.2 Thread Priority*)
- Dependency on a load instruction that missed the L1 Dcache causes a flush and dispatch stall
- Dependency on a store conditional instruction that has not updated the CR causes a flush and dispatch stall
- A load or store instruction D-ERAT miss causes a flush and dispatch stall
- The other thread is executing a microcoded instruction
- A System-Caused interrupt is pending
- A forward progress timer timeout has occurred
- A caching-inhibited load instruction issued with Precise Machine Check interrupts enabled causes a flush and dispatch stall
- Thermal throttling is active

Dispatch Stalling Nop Instructions

A new form of the **nop** instruction allows finer-grained thread control for long latency FPU and VXU instructions. This **nop** is used to stall a thread after a long latency instruction so that the other thread can get all of the dispatch slots.

All instructions for the same thread stall at dispatch for a programmable number of cycles beginning three cycles after the special **nop** instruction is dispatched (i.e. up to two groups can dispatch before stalling begins). This allows the other thread to continue dispatching during this time. This method yields better performance than simply stalling at the VSU Issue Queue (VQ8) stall point since such a stall affects both threads. The instruction format, including the number of stall cycles, is defined in *Section 11.5 Nop Forms of the OR/ORI Instructions* on page 110.

Dependency on a Load Instruction Causing a DCache Miss

When an instruction is detected that has a dependency on an older load instruction that incurs a DCache miss, the dependent instruction and all younger instructions for the same thread are flushed (10 cycles after the dependent instruction is issued from the IS2 point). The flushed instructions are refetched. Dispatch for the flushing thread is then stalled until the load data is available in the DCache.

Dependency on a Store Conditional Instruction

If an instruction is issued with a dependency on a **stdcx.** or **stwcx.** instruction that has not yet updated the CR, then the dependent instruction and all younger instructions for the same thread are flushed (10 cycles after the dependent instruction is issued from the IS2 point). The flushed instructions are refetched. Dispatch for the flushing thread is then stalled until the CR is updated.

Load or Store Instruction D-ERAT Miss

When a load/store instruction causes a D-ERAT miss, all younger instructions for the same thread are flushed once the instruction causing the miss is 10 cycles past the IS2 issue point. The flushed instructions are refetched. Dispatch for the flushing thread is then stalled until the load data is available in the D-ERAT.

Only one outstanding D-ERAT miss from either thread is allowed. If a subsequent D-ERAT miss occurs for the other thread, both the instruction causing the miss and all younger instructions for the same thread are flushed. Dispatch is then stalled for both threads until the first D-ERAT miss is resolved.

Microcoded Instruction

The microcode engine is shared between both threads. When a microcoded instruction is dispatched, both threads are stalled at dispatch until the microcode sequence is complete. The microcode sequence is complete when the last microcode operation is in the ID1 stage. In the following cycle, an instruction from the other (non-microcode) thread can be dispatched and placed into ID1. Therefore, invoking microcode induces a performance penalty on both threads.

System-Caused Interrupt Pending

When a System-Caused interrupt is pending and not masked, dispatch for both threads is stalled until the interrupt is masked or serviced. The processor waits for the current instructions that are already committed to finish. Once it has been determined whether the committed instructions take an exception which masks the system-caused interrupt, or the instructions finish cleanly and the system-caused interrupt is taken, the dispatch stall is removed.

Forward Progress Timer Time-out

A Forward Progress Timer prevents one thread from "starving" the other thread of execution cycles. Since any IS2 stall causes dispatch for both threads to stall, it is often the case that one thread may miss its opportunity to dispatch (see *Section 6.6.3.2 Thread Priority*) because the other thread is stalling at issue. Therefore, in multithreaded mode it is imperative to turn on the Forward Progress Timer in order to avoid thread starvation. There are several registers involved in implementing this facility. They are the Thread Status Register Local (TSRL), the Thread Switch Control Register (TSCR), and the Thread Switch Time-Out Register (TTR) (see *Section 5.2 Privileged register*).

TTR[TTIM] holds the Thread Time-Out Flush Value. The PPU loads the value of TTR[TTIM] into the Forward Progress Timer (TSRL[FWDP]) every time the current thread completes a PowerPC instruction. If the current thread is not suspended (TSRL[TP] != '00'), the Forward Progress Timer is decremented by one each time an instruction completes on the opposite thread.

If TSCR[FPCF] = '1' and the timer reaches x'00 001', then after the next instruction completes, the instruction for the opposite thread is flushed when it reaches the EX7 execution stage. It is re-fetched, and then stalled at dispatch until an instruction completes on the current thread.

When the TSR count becomes ‘00001’, the logic records that a TSR timeout condition occurs. The TSR counter continues to decrement while committed instructions of the opposite thread complete wrapping from ‘00000’ to ‘FFFFF’. However, when the already committed instructions complete, the opposite thread is flushed and dispatch is blocked until the starved thread completes an instruction.

Caching-Inhibited Load Issued with Precise Machine Checks Interrupts Enabled

If Precise Machine Check interrupts are enabled (HID0[en_prec_mchk] = '1') and a caching-inhibited (I = '1') load instruction is issued, then all younger instructions for the same thread issuing the load are flushed, refetched, and then stalled at dispatch. The stall is released when the caching-inhibited load instruction is completed by the PPSS.

Thermal Throttling

The processor can detect and signal a thermal throttling condition if the chip is overheating or to save power. When signaled, dispatch is blocked for one or both threads.

6.6.3.2. Thread Priority

The processor defines three priority levels for a thread: low, medium, and high. At the dispatch stage (IB2), the processor selects which thread to dispatch instructions from using this three-level thread priority. After it selects the thread, the processor dispatches instructions from this thread provided that dispatch for this thread is not otherwise stalled.

Table 6-1 and *Figure 6-13* describe how the processor selects a thread to dispatch based on the thread priority and how the priority between threads is determined.

When deciding which thread can dispatch an instruction, there are three considerations: which thread has highest priority to dispatch, which threads are allowed to dispatch, and whether dispatch is stalled.
In most thread priority modes, the thread that has highest priority is selected to dispatch first. In some modes (that is, medium/medium and high/high), highest priority to dispatch is toggled every cycle that dispatch is not stalled. In contrast, the thread that is chosen to dispatch is not gated by dispatch being stalled. Hence, it is important to note that a thread loses its dispatch opportunity if dispatch is stalled for that thread at the time the thread is allowed to dispatch (for example if the IS2 issue point is stalled or a microcode routine is running). This means that even though the thread dispatch priority policy may be in place, one thread may still prevent the other thread from dispatching because it is stalling. Software should use the Forward Progress Timeout facility to ensure both threads always make forward progress.

Table 6-1. Thread Priority

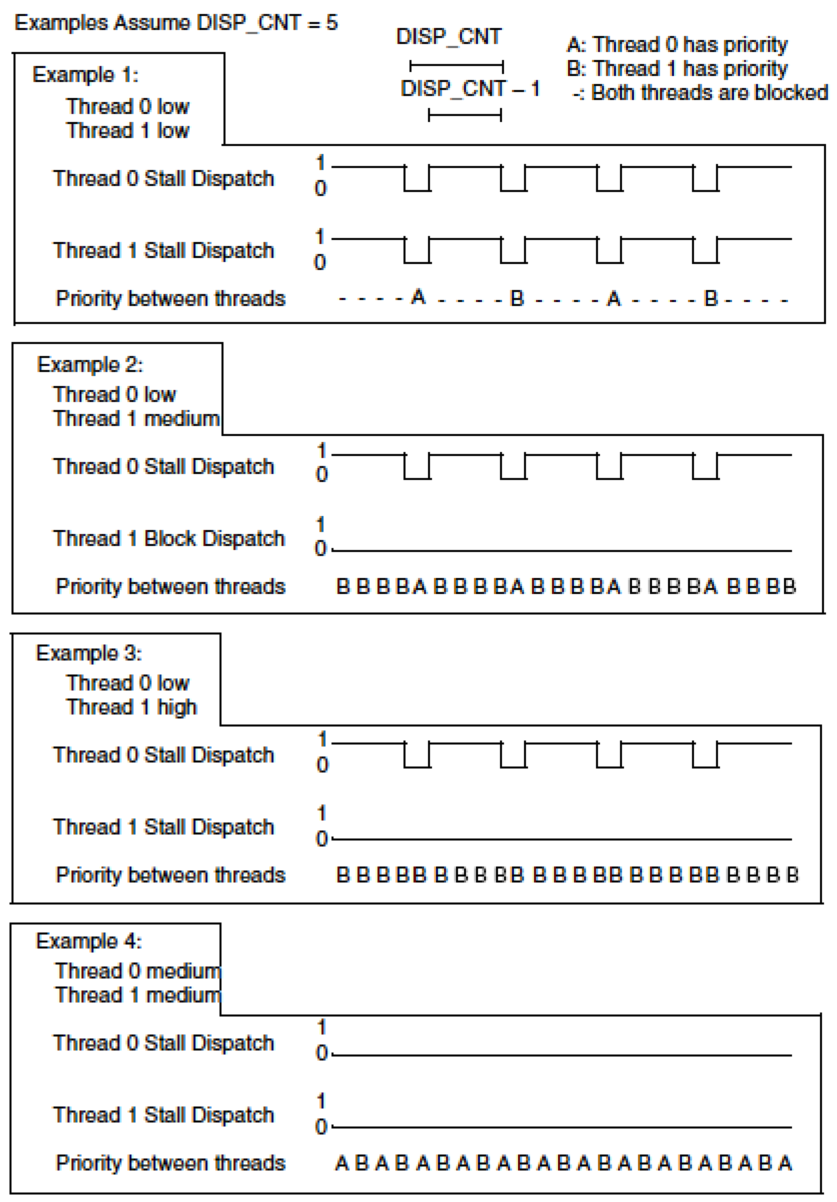
Thread 0 Priority	Thread 1 Priority	Description
Low	Low	Both threads alternate dispatching. If the thread with priority in this cycle has no instructions to dispatch, the other thread can dispatch. Dispatch can only occur once every TSCR[DISP_CNT] cycles.
Low	Medium	In one dispatch cycle, the lower-priority thread is allowed to dispatch during the cycle it has priority. In the other TSCR[DISP_CNT] - 1 cycles, only the higher-priority thread is allowed to dispatch.
Medium	Low	
Medium	High	
High	Medium	
Low	High	The higher-priority thread always has priority to dispatch. One cycle every TSCR[DISP_CNT] cycles, the lower-priority thread is allowed to dispatch. The remaining TSCR[DISP_CNT] -1 cycles, the lower-priority thread is not allowed to dispatch.
High	Low	
Medium	Medium	Even Priority. Both threads alternate dispatch every other cycle. If dispatch is stalled, then priority does not toggle.
High	High	

Note:

TSCR[DISP_CNT] indicates the dispatch count. There are two special cases:

1. TSCR[DISP_CNT] = '1'. Dispatch is not stalled for any combination of thread priorities, and priority alternates between threads every other cycle if the thread priorities are equal. If one thread has higher priority, it always gets the dispatch cycle if it can use it. Otherwise the lower-priority thread gets the dispatch cycle.
2. TSCR[DISP_CNT] = '0'. The dispatch count is 32, which is the maximum value.

Figure 6-13. Thread Priorities Examples



The following events may change the thread priority:

- A **mtspr** instruction writes a new value into the thread priority field (TP) of the TSRL register
- A special priority-changing form of the **nop** instruction is issued
- A System-Caused interrupt occurs

The first two priority-changing events have restrictions on what priority levels can be set for the thread issuing the instruction that depends upon the current executing context of the processor. These conditions are summarized below in *Table 6-2*.

Table 6-2. Thread Priority Table

Condition	Available Thread Priority
Thread in problem state (MSR[PR] = '1') and TSCR[UCP] = '0'	Cannot change priority
Thread in problem state (MSR[PR] = '1') and TSCR[UCP] = '1'	Low and Medium
Thread in privileged state (MSR[PR, HV] = ‘00’) and TSCR[SCP, UCP] = ‘00’	Cannot change priority
Thread in privileged state (MSR[PR, HV] = ‘00’) and TSCR[SCP, UCP] = ‘01’	Low and Medium
Thread in privileged state (MSR[PR, HV] = ‘00’) and TSCR[SCP] = '1'	Low, Medium, and High
Thread in hypervisor state (MSR[PR, HV] = ‘01’)	Low, Medium, and High
Otherwise, the thread priority and the value of TSRL[TP] is not changed.	

Programming Note:

A thread in any state cannot change the priority of the opposite thread.

By using special forms of the **nop** instruction, the priority is changed on completion of the instruction provided the function is enabled for the current privilege level. The following forms are used.

```
'or R1,R1,R1' #Sets the low priority for the active thread
'or R2,R2,R2' #Sets medium priority for the active thread
'or R3,R3,R3' #Sets high priority for the active thread
```

If the following conditions exist for a thread, then the hardware boosts the thread priority level to medium as soon as the interrupt is pending:

- A System-Caused exception occurs,
- The thread priority is low,
- The corresponding interrupt is enabled,
- TSCR[PBUMP] = '1'

This does not change the value in TSRL[TP] for the affected thread. This priority boost remains in effect until software changes the priority.

If a System Reset interrupt is taken for any reason, the thread priority is set to high and the TSRL[TP] field is set to '11'. This is done because if the thread was previously disabled, the value of the TSRL[TP] is ‘00’. When starting the system, the TSRL[TP] needs to be set to something other than '00'. Software can always set the TSRL[TP] to '10' or '01' immediately at address x’0100’ (the System Reset interrupt vector).

Programming Note:

A pending External interrupt will have boosted priority when LPES[0] = '0' and MSR[HV]='0', regardless of the MSR[EE] because it is not being masked. However, a Mediated External interrupt is always masked by MSR[EE], so the priority will NOT be bumped until MSR[EE] = '1'. The general rules always apply, if the interrupt is masked, no priority bump, if it is pending to be taken, then there is a priority bump.

6.6.3.3. Resuming and Suspending a Thread

In contrast to adjusting thread priorities, it is possible to completely suspend a thread. When a thread is suspended, it does not fetch instructions. One thread is active and one thread is suspended when powering up the processor because it makes the boot sequence simpler by having only one thread fetching instructions from memory. Suspending both threads is useful in that it can significantly reduce the power consumption of the processor because the clocks can be shut off if neither thread is active. Shutting off clocks is done by a unit external to the PPE, and is therefore beyond the scope of this document.

Suspending a Thread

There is only one way for software to suspend a thread. Software uses the **mtctrl** instruction to suspend the thread in which the instruction is issued by clearing the corresponding Thread Enable bit in the CTRL register. This can only be done in hypervisor state (MSR[PR, HV] = '01'). In any state, setting the Thread Enable bit to '0' for the opposite thread is ignored. If one thread remains active while the other thread suspends itself, then suspension of the other thread is viewed as a context-synchronizing event to the thread remaining active.

Resuming a Thread

This section describes the process of starting a thread that has never run before or resuming a thread that has previously been suspended. At power-on reset (POR) both threads are suspended. After POR, thread 0 is activated by a System Reset exception. Software must explicitly then resume thread 1. If a thread has been previously suspended, then it can be resumed by any of the following events:

- In hypervisor mode (MSR[PR, HV] = '01'), software can resume the opposite thread from the one that is executing by setting the corresponding thread enable (TE) bit of the CTRL register using the **mtctrl** instruction.
- A Thermal Management interrupt exception occurs and HID0[therm_wakeup] = '1'
- A System Error exception occurs and HID0[syserr_wakeup] = '1'
- An External exception occurs and TSCR[WEXT] = '1'
- A Decrementer exception occurs for the suspended thread and TSCR[WDEC0] or TSCR[WDEC1] = '1' for the same corresponding thread. The decrementer for a suspended thread continues to run while the thread is suspended.

A Machine Check exception cannot occur for a suspended thread. Also, a Hypervisor Decrementer exception does not resume a thread. In the event of a System-Caused exception, the thread resumes even when the corresponding interrupt is masked. At POR, HID0[therm_wakeup, syserr_wakeup] = '00', TSCR[WDEC0, WDEC1] = '00', and TSCR[WEXT] = '0'. This disables waking up a suspended thread on a corresponding exception.

A resumed thread starts execution from the System Reset interrupt vector location. The thread priority is set to high and TSRL[TP] field is set to '11' by hardware independently of how the suspended thread has been resumed. When a resumed thread starts executing at the System Reset interrupt vector location, SRR1[42:44] is set to a thread resume reason described in *Table 6-3 Thread Resume Reason* . If multiple reasons are present for the thread to resume, then priority of the interrupts is indeterminate, but a valid reason will be specified. Additionally, in the time lag between waking up the thread and enabling interrupts, additional exceptions can occur. The interrupt taken is based on the interrupt priorities at the time the interrupt is taken, not on what caused the thread to resume.

The remaining SRR1 fields contain the value from the MSR as if the thread had never been suspended. SRR0 similarly contains the address of the instruction that would have been executed if the thread had not been suspended. The System Reset interrupt handler can use this information to resume the suspended thread properly. Since a System Reset thread resume and a **mtctrl** thread resume have the same reason code, software must look at the value of HID0 or HID1 to determine which of these caused the resume. Since a System Reset can only occur at POR, if the HID registers are set to their POR value, then the thread resume must be the result of a System Reset. If the value of the HID registers differ from their POR value, then the thread resume must be the result of a **mtctrl** instruction.

Table 6-3. Thread Resume Reason Code

Thread Resume Reason	SRR1[42:44]
System Reset exception	‘101’
mtctrl	‘101’
Thermal Management interrupt exception	‘010’
System Error exception	‘110’
External interrupt exception	‘100’
Decrementer interrupt exception	‘011’

When a thread is resumed by a Thermal Management, System Error, External, or Decrementer exception, the resuming interrupt is then disabled because MSR[EE, HV] will be set to '01' when the resuming thread is started at the System Reset vector. If the interrupt is later enabled by changing the MSR settings, and the exception is still pending, then the resuming interrupt will finally be taken. In other words, a resuming interrupt causes a suspended thread to resume, but does not cause the resumed thread to take the resuming interrupt until interrupts are explicitly enabled by the resumed thread.

When two threads are resumed in exactly the same clock cycle (a rare event), it is indeterminate which thread is resumed first. However, both threads resume properly.

When one thread is active and the other is inactive and the inactive thread is resumed, the active thread views the inactive thread resuming as a context-synchronizing event.

6.6.4. Exceptions and Multithreading

This section summarizes the multithreading behavior for the exceptions defined in *Section 9 Exceptions and Interrupts* on page 80.

- External, Precise Machine Check, System Reset, Thermal Management, and System Error exceptions are thread dependent and handled by the thread that causes the exception. Each of these interrupts is enabled for both threads by TSCR[WEXT], HID0[en_prec_mchk], HID0[en_syserr], HID0[therm_wakeup], and HID0[syserr_wakeup] respectively (see *Section 6.6.3.3 Resuming a Thread* on page 53).
- Decrementer exception
  Each thread has a separate Decrementer register that creates a separate Decrementer interrupt for each thread. TSCR[WDEC0] or TSCR[WDEC1] enables waking up a suspended thread due to a Decrementer Interrupt (see *Section 6.6.3.3 Resuming a Thread* on page 53).
- Hypervisor Decrementer exception
  A single Hypervisor Decrementer register is shared between both threads. A Hypervisor Decrementer exception does not resume a suspended thread.
- Imprecise Machine Check exception
  Imprecise Machine Check interrupts do not have thread information associated with them. All enabled threads take an Imprecise Machine Check interrupt regardless of which thread caused the exception. An Imprecise Machine Check exception does not resume a suspended thread. If the thread is later enabled, it does not take an Imprecise Machine Check interrupt.
- Maintenance exception
  An Instruction-Caused Maintenance interrupt is thread dependent. It is handled by the thread that caused the exception. A System-Caused Maintenance interrupt is shared between threads and occurs when one or both of the two trace input signals, which are shared by both threads, are asserted. The System-Caused Maintenance interrupt is handled by the thread that enables the interrupt.
- Floating-point enabled exceptions
  When either floating-point exception mode bit (MSR[FE0], MSR[FE1]) is set, and a floating-point operation is executing, the PPU switches into a single instruction issue mode. In this mode only one instruction is active in execute at a time. All other instructions from either thread are held at issue until the previous instruction completes.
- Other Exceptions
  All other exceptions are thread-dependent and are handled by the thread that caused the exception.

7. PowerPC Processor Storage Subsystem (PPSS)

The PowerPC Processor Storage Subsystem (PPSS) deals primarily with moving instructions and data between the PPU, the L2 Cache, and the Element Interconnect Bus (EIB). For a high-level view of the PPSS see *Figure 2-1 PPE High-Level Block Diagram* on page 7.

In this chapter, the various subunits of the PPSS are presented in high-level detail.

7.1. PPSS Implementation Subunits

The PPSS implementation is divided into four major subunits:

1. Core Interface Unit (CIU)
   - Eight entries store request queues for the PPU
   - One entry store queue for the MMU
   - Four entries for the demand load. The demand load does not require prefetching. It is a regular load request from the Load Store Unit (LSU) that results from a load instruction.
   - One entry for the instruction demand fetch for each thread (instruction fetch only, no prefetch)
   - Data prefetch engine
   - 32-byte reload data bus
2. Noncacheable Unit (NCU) and the line buffer
   - Four 64-byte store gathering
   - Four store queue entries
   - NCU store pipelined
3. L2 cache
   - Eight-way, 512 KB, copy back, 128-byte line
   - Eight 64-bytes store gathering queue
   - Six read-and-claim (RC) machines
   - Locking feature through the L2 replacement management table (RMT)
   - Data is 128 bits ECC protected
   - Redundant directories for the PowerPC Processor Element and the BIU snoop traffic for parity error handling
4. PPE Bus Interface Unit (BIU) to the internal EIB

7.2. PPE Core Interface Unit (CIU)

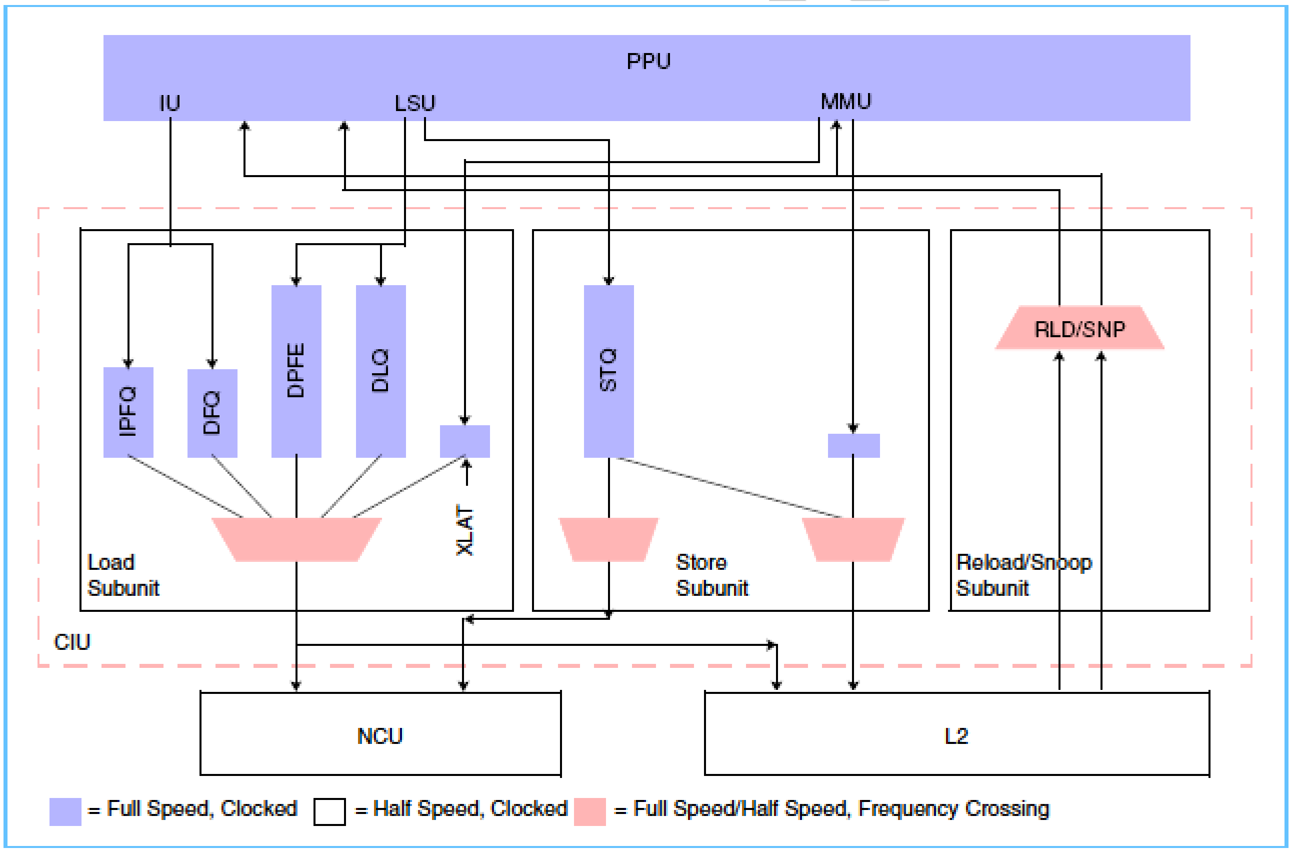
The Core Interface Unit (CIU), located on the boundary of the PPSS and the PPU. It functions as a routing, arbitration, and flow control point for requests originating from the Load Store Unit (LSU), Instruction Unit (IU), and Memory Management Unit (MMU) destined for the Level 2 Cache Unit (L2) and Noncacheable Unit (NCU) units, as well as the subsequent responses and requests originating from the L2 and NCU destined for the LSU, IU, and MMU. The PPU operates at full frequency; whereas the L2 and the NCU are designed for half-frequency operation. The order between load and store requests is enforced by the CIU for both cacheable (I='0') and caching-inhibited (I='1') addresses when the addresses of the requests are to the same 128-byte cache block.

The CIU is comprised of three major functional subunits:

- Load – representing instruction and data requests from the LSU, IU, and MMU as well as software data-prefetch
- Store – representing memory updates sent from the LSU and MMU
- Reload – representing load request responses from the L2 or NCU to the LSU, IU, and MMU

In addition, the software data-prefetch function (**dcbt** instruction) is implemented in the CIU and is part of the Load subunit. With the exception of the load-hit-store comparison, none of the subunits share any signals and each works independently of the others. *Figure 7-1* shows a high-level diagram of the CIU subunits.

Figure 7-1. CIU High-Level Block Diagram

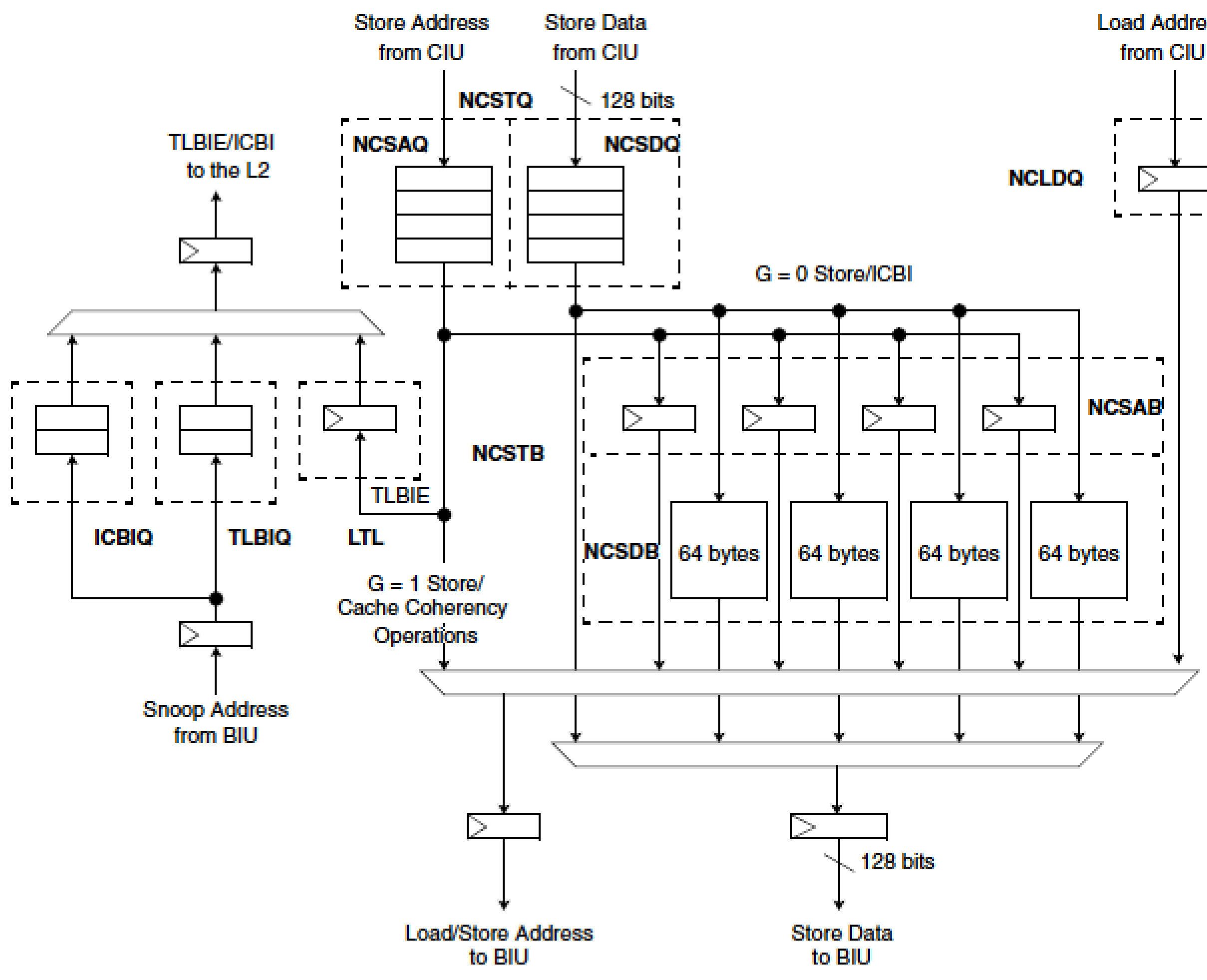


XLAT	MMU translate request queue
DLQ	demand data load request queue
DPFE	Data Prefetch Engine (supports eight software-managed data prefetch streams)
DFQ	demand instruction request queue
IPFQ	instruction prefetch request queue
STQ	store queue

7.3. Noncacheable Unit

The noncacheable unit (NCU) is a subunit of the PowerPC Processor Storage Subsystem. The main function of the NCU is queueing/buffering of noncacheable operations between the PPU and the memory system. The NCU interfaces with the CIU, the L2 cache unit, and the Bus Interface Unit (BIU).
A dataflow diagram of NCU is shown in *Figure 7-2*.

Figure 7-2. Dataflow Diagram of NCU



BIU	Bus Interface Unit (BIU)	NCSAQ	Noncacheable store address queue
CIU	Core Interface Unit	NCSDB	Noncacheable store data buffer
ICBIQ	Instruction cache block invalidate queue	NCSDQ	Noncacheable store data queue
LTL	Locally-originated **tlbie(l)** latch	NCSTB	Noncacheable store buffer
NCLDQ	Noncacheable load queue	NCSTQ	Noncacheable store queue
NCSAB	Noncacheable store address buffer	TLBIQ	TLB invalidate entry queue

The NCU has following properties:

- Handles Caching-Inhibited (I='1') fetch/load/store operations and several cache coherency operations
- Supports one Caching-Inhibited (I='1') fetch/load at a time
- Provides four 64-byte store-gathering buffers (NCSDB for nonguarded (G = '0') stores)

The NCU maintains all access ordering for operations accessing the same addresses (at 16-byte granularity). It also maintains caching-inhibited, guarded (IG ='11') store ordering regardless of address.

In addition to handling all caching-inhibited (I = '1') accesses, the NCU handles the following instructions: **dcbz** (I = '1' only), **eieio**, **sync** (L = 0, 1, or 2), **icbi**, **tlbie**(**l**), and **tlbsync**.
The **tlbsync** instruction is treated as a **nop** in the NCU because the implementation is self-synchronizing around the **tlbie**(**l**) instruction itself.

The lightweight **sync** (L = 1) is completed in the NCU and is not sent to the EIB. Depending on the MMIO settings, this behavior can be changed to map the **sync** (L = 1) into a EIEIO, or SYNC type of bus operation. If changed to a SYNC bus operation the NCU does not wait for a "sync done" signal as it would for the **sync** (L = 0) instruction.

The NCU consists of following queues and buffers (see *Figure 7-2*):

- Noncacheable load queue (NCLDQ)
- Noncacheable store queue (NCSTQ)
  - Noncacheable store address queue (NCSAQ)
  - Noncacheable store data queue (NCSDQ)
- Noncacheable store buffer (NCSTB)
  - Noncacheable store address buffer (NCSAB)
  - Noncacheable store data buffer (NCSDB)
- Remotely-originated **tlbie** queue (TLBIQ)
- **icbi** queue (ICBIQ)
- Locally-originated **tlbie**(**l**) Latch (LTL)

7.4. L2 Cache

The L2 Controller handles all cacheable (I = '0') loads and stores (including **lwarx**/**ldarx**/**stwcx.**/**stdcx.** instructions), data prefetches, instruction fetches, instruction prefetches, cache operations, and barrier operations. The following sections describe the features of the L2 function.

7.4.1. L2 Cache Controller Features

The L2 cache controller has the following features:

- Critical quadword-forwarding on data loads and instruction fetches
- Six reload queues and six castout queues
- Eight 64-byte wide store queues
- Store-gathering
- Nonblocking L1 Data Cache invalidates
- Recoverable single-bit directory errors (sourced from redundant directory)
- Least Recently Used (LRU) algorithm with support for replacement management (i.e. cache-locking)
- Multithread support via two atomic reservation stations
- Global and dynamic power management
- Separate snoop directory for all system bus snoops
- Four snoop intervention/push queues
- Error Correction Code (ECC) for data-protection

7.4.2. L2 Storage Characteristics

Table 7-1 gives a high-level overview of L2 storage characteristics.

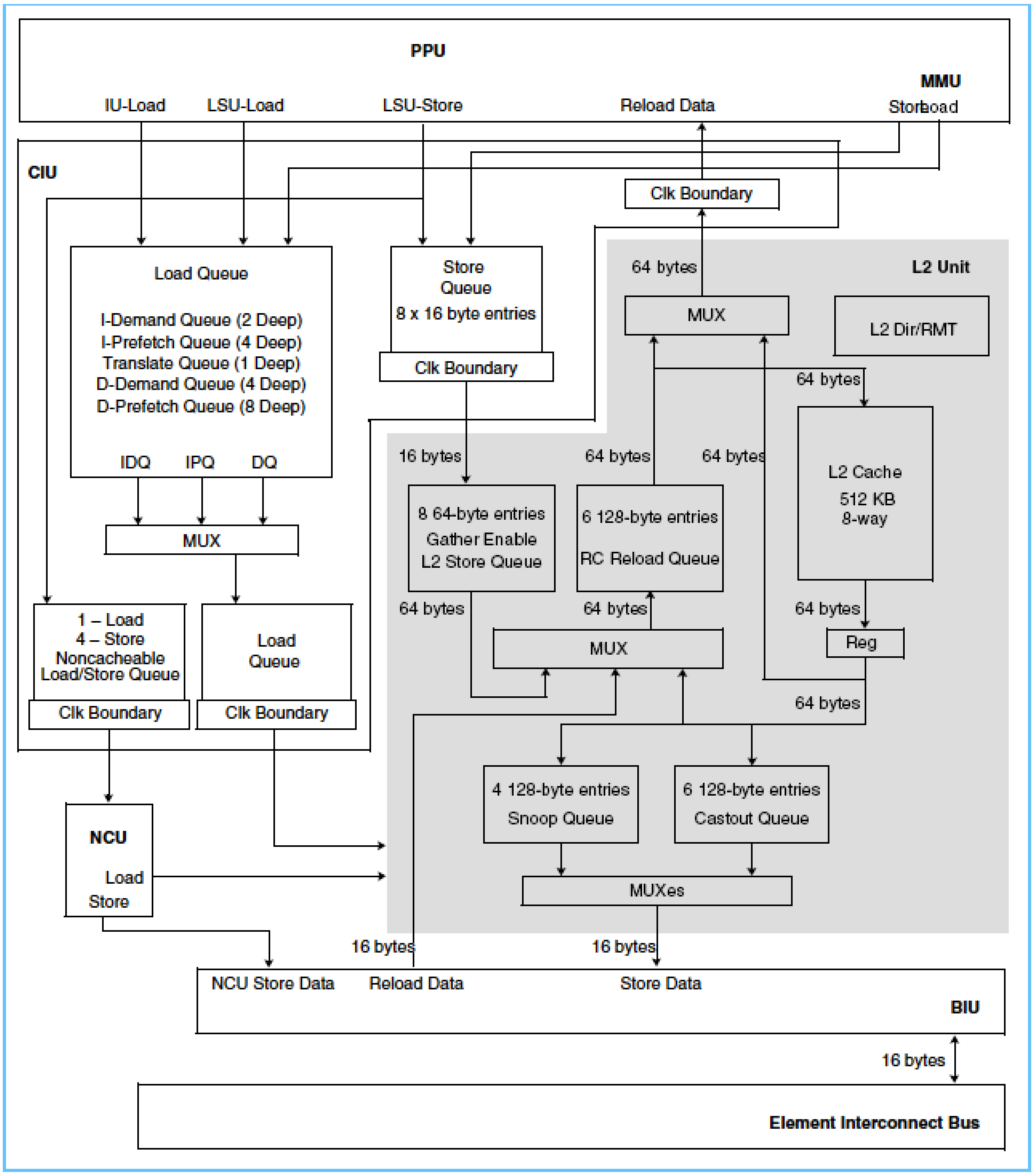
Table 7-1. Storage Characteristics Summary

Characteristic	L2 Cache Parameters
Data type	Shared (both instruction and data)
Size	512 KB
Associativity (replacement policy)	Eight-way set associate (or direct-mapped via the L2_ModeSetup1[63] register)
Line size (sector)	128 bytes
Operation granularity	128 bytes
Index	Real address
Tags	Real address
Number of ports	One read port or one write port
Inclusivity	Inclusive of L1 data Not inclusive of L1 instructions (software must use **icbi** to maintain coherency)
Hardware coherency	Supported. A separates directory array is maintained for snoops
Store policy	Write-back. Allocation is done at the time a store miss occurs
L2 Bypass Control	Use HID4[dis_force_ci, enb_force_ci] to bypass the L2 (and L1).
Reliability, Availability, Serviceability (RAS)	ECC on data; parity on directory tags (recoverable using redundant directories)

7.4.3. PPSS L2 High-Level Structure

Figure 7-3 shows the high-level structure of the PPSS L2 cache.

Figure 7-3. PPSS L2 High-Level Structure



7.4.4. L2 Cache Management

The L2 is a unified cache that maintains full cache-line coherency within the system and can supply intervention data to other processor units. Logically, the L2 is an in-line cache. Unlike the write-through L1 caches, the L2 is a write-back cache. It is fully inclusive of the L1 DCache, but is not inclusive of the L1 ICache.

The size of the L2 is a total of 512 KB (with 4-KB cache lines). It is physically implemented as four L2 quadrants, each quadrant containing 32 bytes of the 128-byte cache line.
A cache-line read from the L2 array requires two data beats (a beat is a one L2 clock cycle, which is two PPU clock cycles), 16 bytes of data from each quadrant per beat. The L2 then sends two 64-byte beats to the CIU data reload bus.

A cache-line read request that misses the L2 is sent to the EIB (through the BIU). Reload data coming from the EIB is received in eight back-to-back 16-byte data beats.
Because the core requires that all reload data must be sent back-to-back, the L2 must wait for all missed data from the EIB before sending data to the core. The L2 allows the core to request which quadword it wants to receive first from the L2 (that is, the "critical" quadword).

A cache-line write requires two write accesses to the L2 array, performing a 64-byte write for each access.

L2 Store Queue

The L2 logic has a queue dedicated for cacheable stores. The queue consists of eight fully associative 64-byte sectors. Each sector is byte writable, so gathering occurs at the point the buffer is written. This is effectively a store cache.

Store Performance Enhancement

If the store queue detects and sends a full line store indication with its request, the RC state machines handle back-to-back writes to the same cache line. This completes a cache line write in 13 cycles (12 cycles for the normal 64-byte write, plus one cycle for the adjacent 64-byte data).

Read-and-Claim Queues

The six RC state machines and their corresponding address and data queues manipulate data in and out of the L2 cache in response to processor requests.

Castout Queues

A line is said to be "castout" when it is removed from a cache. The line must be coherent before it can be removed. The six castout (CO) state machines and their corresponding address and data queues are closely tied (one-to-one) to the six RC state machines.

Snoop Queues

Snoop state machines, with their corresponding address and data latches, accept snoop requests from the Bus Interface Unit (BIU) and perform or request the necessary transactions to keep the L2 Cache coherent with the overall system.

L2 Cache States (MERSI, Mu, and T)

The L2 Cache uses the MERSI cache coherency protocol plus two additional states (Mu and T).
(M: Modified, E: Exclusive, R: Recent, S: Shared, I: Invalid, Mu: Unsolicited Modified, T: Tagged)

7.4.5. L2 Replacement Algorithm

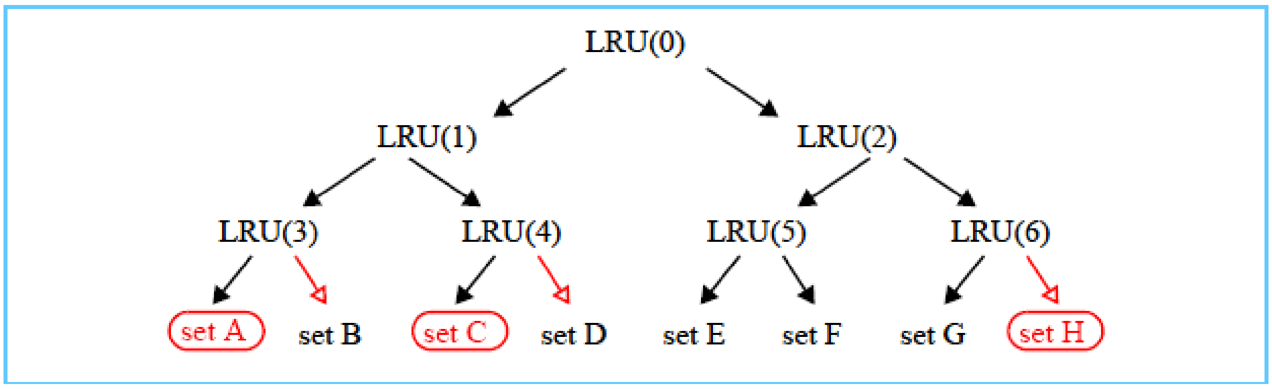
The PPSS L2 cache replacement algorithm can be operated in one of three modes: Binary Least Recently Used (LRU), pseudo LRU (p-LRU), and direct mapped.
The Binary LRU algorithm employed is based on the binary tree scheme.
The p-LRU method uses configurable address registers and the MMIO Replacement Management Table (RMT) to lock one to eight cache partitions to a particular RclassID.
The direct mapped mode uses three tag address bits to map an address to one of the eight congruence class members.

7.4.5.1. Pseudo LRU Mode (using RMT)

The L2 LRU supports the following features:

- Multiple cache lines (sets) locking
  In the PPSS, the L2 LRU replacement requirement is used to perform cache line replacement with the normal LRU method and with multiple cache lines (sets) that are locked.
- RclassID from PPU to LRU controller
  The LRU controller uses the RclassID from the PPU to determine which set or sets are locked.
- Cache line replacement involves the RMT in the LRU controller.
  The LRU controller must look up the LRU value for the corresponding cache line and then update the LRU value based on the look-up value from the LRUs eight 8-bit RMTs using the 3-bit subclass ID as the index. If the value of the entry in the RMT is '0', the corresponding set is locked. When the value in the table is ‘1’, the corresponding set can be chosen for replacement during a miss case.
  If the RMT data contains all zeros, then the entry is treated as all ones with no locked sets.
- Updated LRU data (pointer) for unlocked cache lines
  The LRU controller also must update the least recently-used pointer to the next unlocked cache line.

Figure 7-4. LRU Binary Tree with Locked Sets A, C, and H



In the example shown in *Figure 7-4*, if the RMT entry return set replacement value is '0101 1110', then:

1. Sets A, C, and H are locked from replacement.
2. The RMT value overrides the following LRU data:
   lru(3) = '1', lru(4) = '1', lru(6) = '0' (The right and left arrows correspond to '1' and '0' respectively).
3. The LRU control must apply pseudo LRU replacement only on sets B, D, E, F, and G in a miss case, or if there is a hit on any unlocked set.
4. For a hit on a locked set case, the LRU controller does not perform an LRU update, and only the set that is hit is replaced.

7.4.5.2. L2 Cache Flush Algorithm

The following procedure flushes the entire L2 cache to memory via software. The other thread should be quiesced or guaranteed not to be performing cacheable operations.

1. Set the mode bits to direct-mapped mode (the RMT must be disabled). Switching between LRU mode and direct-mapped mode can be done at any time. Data integrity is maintained, although performance might be affected.
2. Pick a 4-MB addressing region of memory to hit all eight ways on direct-mapped mode.
3. Execute one load, store, or **dcbz** instruction to bytes in each 128-byte cache line in the chosen memory region. This fills the L2 with the contents of the chosen region and forces out all other data contained in the cache.
   Note: At minimum, for each value of addr[42,43,44], it is required to cycle through all values of addr[48:56]. This accesses a 64-KB region for each way, for a total of 512KB.
4. Execute one **dcbf** instruction for each 128-byte cache line in the specified memory region. This flushes out the specified region to the cache. If the contents of the specified region were resident and modified in any L2, this flush ensures that the data iSPUshed out to memory.

Note: The other thread should be quiesced or guaranteed not to be performing cacheable operations.

7.4.5.3. Reservation (LARX/STCX)

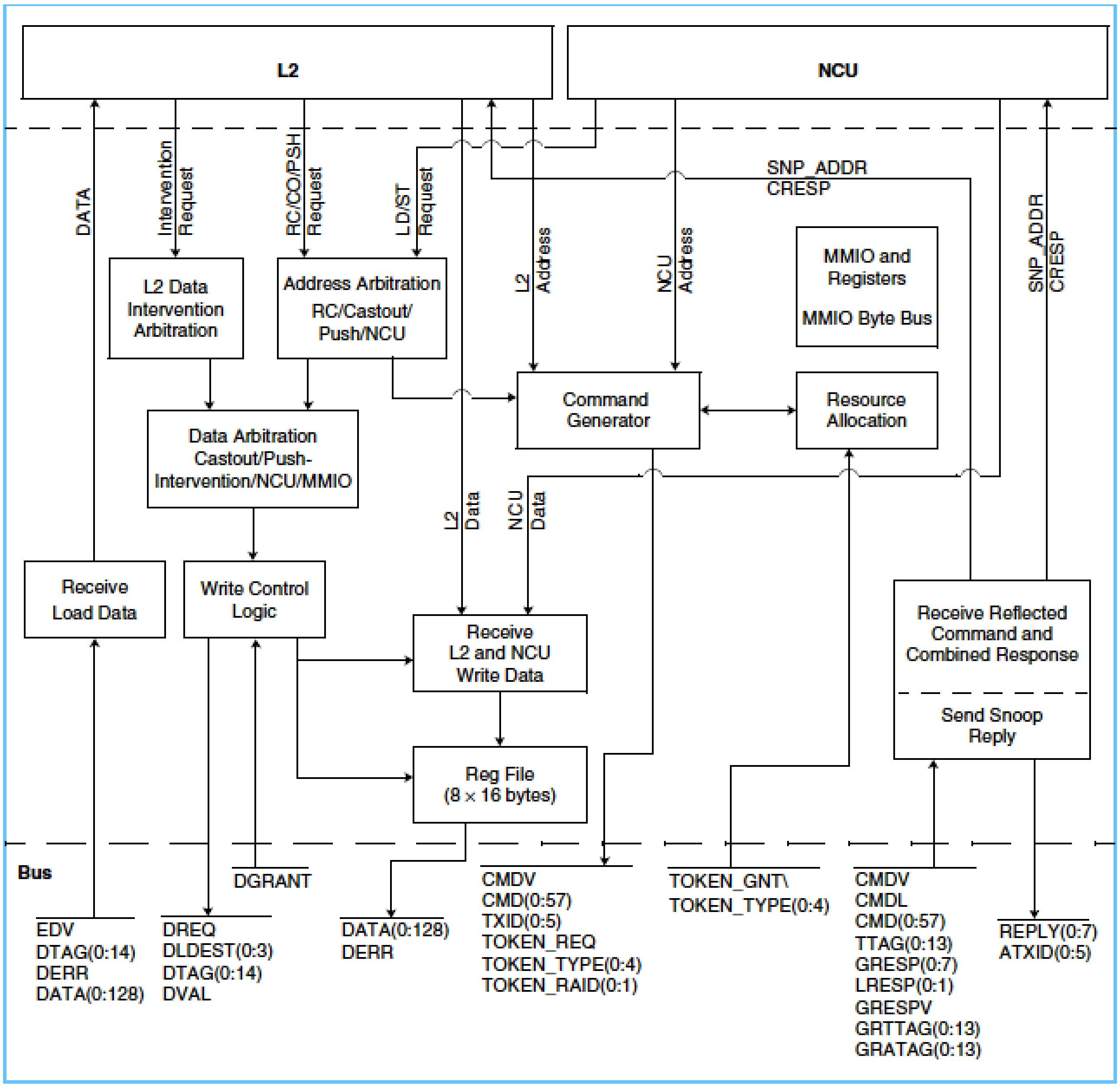
The L2 contains logic to support two reservation bits (one per thread) to handle a LARX request by each thread. The Snoop logic compares against both reservation addresses. This is not a thread-based comparison.

7.5. PPE Bus Interface Unit

The PPE bus interface unit (BIU) provides an interface between the EIB, Token Manager (TKM) (in resource-allocation enabled mode), and the L2/NCU units of the PPE. The BIU is also the master on the PowerPC Processor Storage Subsystem (PPSS) internal MMIO area.

Figure 7-5 is a high level block diagram of the BIU.

Figure 7-5. High-Level BIU Block Diagram



7.5.1. Features

The BIU has the following features:

- Is a master device and a slave device on the EIB, architected to support fully coherent memory operations
- Implements a credit-based flow control facility for commands that limits the total number of commands that can be sent to the EIB.
- All data operations on the EIB are assumed to take eight beats. It is designed around 128-byte cache lines, and the coherency and synchronization granularity is 128 bytes.
- 16-byte data-in interface and 16-byte data-out interface to the EIB (+1 reserved bit)
- 16-byte data-in interface and 16-byte data-out interface to the L2 (+2 reserved bits)
- 16-byte data-in interface from NCU (for NCU data loads, the data-out interface to the L2 is used)
- Supports incoming MMIO load/store commands initiated by the input/output controller (IOC). Acts as a master device on the PPSS internal MMIO area. The PPSS MMIO area is comprised of the BIU, the L2, the CIU, and the NCU.

The BIU supports two modes of operation — resource allocation enabled mode and resource allocation disabled mode. The default mode is resource allocation disabled mode.

As a master, the BIU sources load/store requests to the EIB for service for the L2 and the NCU.

- Request and Command (Address Phase) Arbitration
  The address phase arbitration logic implements a three-stage round robin arbiter to arbitrate between the L2 and the NCU load/store requests, and selects the winner.
- Sending Requests and Commands to the EIB
  If the BIU has EIB-issued credits, the command-formatter/dispatch logic sends the command to the EIB. To support the resource allocation, the command-formatter/dispatch logic also checks if the command to be sent requires a token and if the corresponding token is available. Only then it can send the command to EIB.
- Sending Data to the EIB
  The BIU implements a high-bandwidth data interface capable of sending data to the bus at a sustained 16 bytes/cycle for each 128-byte packet. All data transfers have a maximum size of 128 bytes and require eight beats of 16 bytes to transfer a 128 byte line. All L2 (castout, push, and intervention) initiated data transfers are 128 bytes. All NCU stores and MMIO are 1-beat transfers on the EIB. After receiving the combined response of acknowledge with no retry, the BIU initiates a data transfer based on the result of a previously received command request.
- Reissuing Retried Commands to the EIB
  BIU implements a pseudo-random delay function to determine the duration of the delay before the command is reissued. The delay is based on the number of times the command has already been retried.
- Receiving Reflected Commands from the EIB
- Sending Snoop Replies to the EIB
- Receiving Local/Global Combined Responses from the EIB
- Receiving Data from the EIB

8. Memory Management

8.1. Address Translation Overview

Please refer to the following documents to understand the Address Translation mechanism.

"4.3 Address Translation Overview" of *Book III: PowerPC Operating Environment Architecture, Version 2.02*
"4.4 Virtual Address Generation" of *Book III: PowerPC Operating Environment Architecture, Version 2.02*
"4.5 Virtual to Real Translation" of *Book III: PowerPC Operating Environment Architecture, Version 2.02*

8.2. Address Translation Implementation

This section describes the address translation facility and storage control features for this processor implementation. This processor implements multiple levels of address translation hierarchy.

All addresses are first translated using a high-speed array; an Instruction Effective-to-Real-Address Translation (I-ERAT) array is used for instruction addresses and a Data Effective-to-Real-Address Translation (D-ERAT) for data addresses.
Note: This translation takes place regardless of the MSR[IR, DR] settings.

If the address is not found in the I-ERAT or D-ERAT (as appropriate), then the address is sent to the Memory Management Unit (MMU) for translation. If the address is in real mode (MSR[IR] = '0' for instructions, MSR[DR] = '0' for data), then the real-mode (non-relocate) translation facility is employed; otherwise, the address is translated by the virtual-mode (relocate) translation facility (see *Section 8.3* through *Section* 8.5).

Virtual-mode translations proceed through a sequence of arrays beginning with the Segment Lookaside Buffer (SLB), followed by the TLB, and finally the Page Table if a matching TLB entry is not found. The process implements the parameters shown in *Table 8-1*.

Table 8-1. Summary of Virtual-Mode Parameters

Parameter	Implemented Value
Effective Address Size	64 bits
Virtual Address Size (n)	65 bits
Real Address Size (m)	42 bits
Large Page Sizes (p)	Two of the following: 64 KB (p = 16) 1 MB (p = 20) 16 MB (p = 24)
D-ERAT Entries	64 shared (2 x 32)
I-ERAT Entries	64 shared (2 x 32)
SLB Entries	64 per thread
TLB Entries	1024 shared

The high-order bits of the virtual and real address are not implemented. The hardware always treats these bits as zero. In other words, VA[0:14] are always zero and RA[2:21] are always zero. Software must not set these bits to any other value than zero or the results are undefined in this implementation.

8.2.1. WIMG Behavior

Four bits in the page table control the processor's accesses to cache and main storage. These are called the WIMG bits, where “W” stands for write through, “I” for cache inhibit, “M” for memory coherence, and “G” for guarded storage.
This processor implementation supports three WIMG settings for page table entries (and TLB entries): ‘0010’, ‘0100’, and ‘0101’. All other settings are implemented as shown in *Table 8-2* (a dash represents either a ‘0’ or a ‘1’).

Table 8-2. Summary of Implemented WIMG Settings

WIMG Setting	WIMG Implementation
- 0 - -	0 0 1 0
- 1 - 0	0 1 0 0
- 1 - 1	0 1 0 1

As *Table 8-2* indicates, if the storage access is cacheable then it is also memory coherent and nonguarded. All storage accesses are performed with the write-through bit set to zero (W = '0').

8.2.2. Data Effective to Real Address Translation (D-ERAT)

The PPE includes a 64-entry, two-way set associative data Effective-to-Real-Address Translation (D-ERAT) cache for fast translation of data effective addresses into real (physical) addresses. Each entry of the D-ERAT contains translation information for a 4KB block of effective storage (even if this section of storage is translated via a large page translation).

No access can bypass the D-ERAT. The D-ERAT must identify each translation entry with MSR[SF, DR, PR, and HV] bits. This allows the D-ERAT to distinguish between translations that are valid for the various modes of operation.

The D-ERAT is shared by both threads, hence all entries are identified by the thread context in which the entry was created. Each thread maintains its own entries in the D-ERAT and cannot use the entries created for the other thread.

Because the content of each virtual-mode D-ERAT entry is the result of a Page Table search based on the contents of an SLB entry, to maintain consistency with the SLB, the following instructions cause all entries in the D-ERAT to be invalidated.

- **slbia** (for entries belonging to the same thread only)
- **tlbie**(**l**) to a large page only (L = '1') (all entries regardless of thread)

The **slbie** instruction causes a thread-based, class-sensitive invalidation of the D-ERAT. That is, it invalidates any entries that have a thread and class-bit match with the thread that issued and class indicated by the **slbie** instruction. In addition, the execution of **tlbie**(**l**) to a small page (L = '0') or the detection of snooped-tlbie operation from another processor causes an index-based invalidate to occur in the D-ERAT. All entries in the D-ERAT that have matching effective address bits [47:51] with register RB[47:51] provided by the instruction are invalidated.

The replacement policy used by the D-ERAT is a simple 1-bit Least Recently Used (LRU) policy.

Each D-ERAT entry is set to the invalid state at power-on-reset (POR).

8.2.3. Instruction Effective to Real Address Translation (I-ERAT)

The PPE includes an 64-entry, two-way set associative instruction Effective-to-Real-Address Translation (I-ERAT) cache for fast translation of instruction effective addresses into real (physical) addresses. The I-ERAT is identical in behavior to the D-ERAT.

8.3. Segment Lookaside Buffer (SLB)

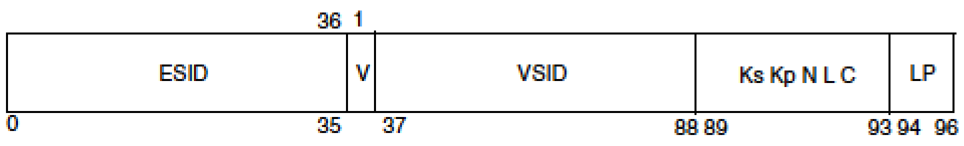
The PPE contains one unified (combined for both instruction and data), 64-entry, fully associative Segment Lookaside Buffer (SLB) per thread. Information derived from the SLB can be cached in the I-ERAT or the D-ERAT along with information from the TLB. As a result, many of the SLB management instructions have effects on the ERATs as well as on the SLB itself.

Because the SLB is managed by the operating system, it is possible that multiple entries may be incorrectly set up to provide translations for the same effective address. This results in undefined behavior and can result in a Checkstop.

SLB Entry Definition

Figure 8-1 shows the SLB Entry definition for this implementation.

Figure 8-1. SLB Entry Definition



The definition of the fields in *Figure 8-1* can be found in the Segment Lookaside Buffer section of *Book III: PowerPC Operating Environment Architecture*. This implementation supports a 65-bit virtual address; hence, SLB Entry[37:51] (or VSID[0:14]) are always zero.
Additionally, this implementation adds the LP field to the SLB Entry (see *Section 8.4.2 Large Page-Size Decode* on page 74). The LP field is shown in *Figure 8-1* to be 3 bits wide in order to be compliant with future versions of the PowerPC Architecture. This processor only implements the low-order bit of the LP field. For this reason, other LP references in this document describe the LP field as a single bit.

8.3.1. SLB Management Instructions and ERAT Coherency

Software maintains the SLB using the **slbmte**, **slbmfee**, **slbmfev**, **slbie**, and **slbia** instructions.
All other optional PowerPC Architecture instructions provided for 32-bit operating systems and for changing the contents of the SLB are NOT implemented in this design (**mtsr**, **mtsrin**, **mfsr**, and **mfsrin**).

The SLB management instructions are implemented as move to or move from SPR instructions in microcode. Because these are microcoded, process performance is degraded when using these instructions. Software should attempt to minimize the number and frequency of these instructions required.

For all SLB management instructions, the first 15 bits [0:14] of the VSID field and the first six bits [52:57] of the index field are not implemented. Writing them is ignored and reading them returns zero.

This implementation adds the LP field (see *Section 8.4.2*) to the RS register definition of the **slbmte** instruction and the RT register definition of the **slbmfev** instruction. In both cases, this field is added as bits [57:59] of the corresponding register. Bits [57:58] are not implemented, so the LP field is treated as a single bit [59].

The **slbie** and **slbia** instructions set the valid bit of the SLB entries that are invalidated to 0. The remaining contents of the SLB entry are unchanged and can be read for purposes of debugging. Therefore, when the hypervisor initiates a process or partition context switch, the hypervisor must set any used SLB entries to zero to prevent establishing a covert communication channel between processes or partitions.

For the effect these instructions have on the ERATs, see *Section 8.2.2 Data Effective to Real Address Translation (D-ERAT)* on page 68.

8.4. Translation Lookaside Buffer (TLB)

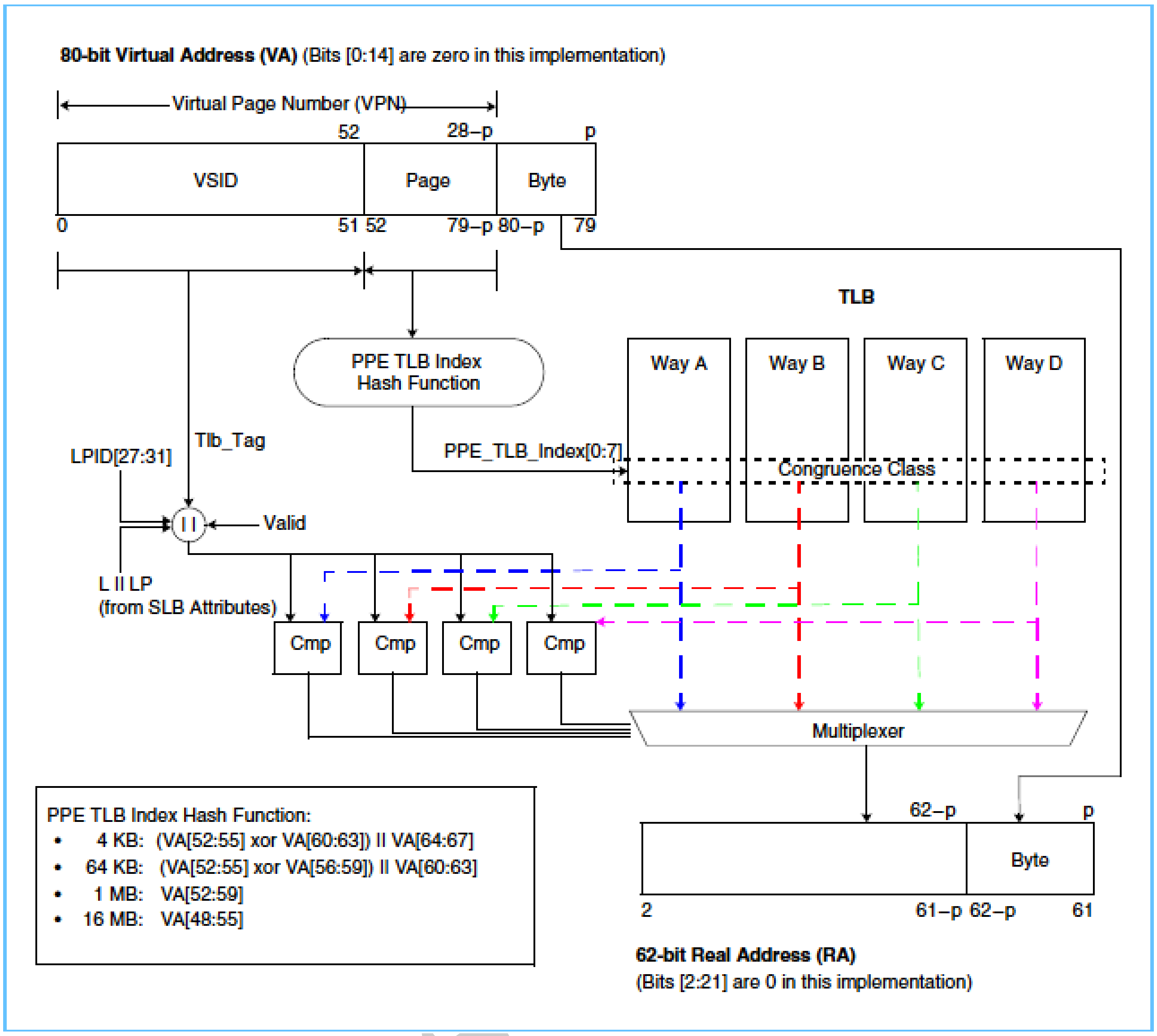
The PPE contains a unified (combined for both instruction and data), 1024-entry, 4-way set associative Translation Lookaside Buffer (TLB) that is shared between threads.

The end result of the SLB lookup operation is a Virtual Address along with Segment Attributes. This information is passed to the TLB to be converted to a Real Address with associated Page Attributes. *Figure 8-2* on page 71 provides a graphical depiction of the address conversion. The Virtual Address is split to form a raw index (which is input to a hash function) and compare tag for the TLB. The result of the index hash function described in *Figure 8-2* produces the TLB Index that chooses among the 256 congruence classes (sets or rows) of the TLB.

The compare tag (see *Table 8-3* on page 71) selects among the four ways (or columns). The compare tag also contains the LPID, Valid (equal to '1'), L (Large Page), and LP (Large Page Selector). Both L and LP come from the SLB Segment Attributes. The compare tag is compared against the data stored in the TLB for each way entry. Note that the width of the compare tag changes for each page size. If a hit results from the compare operation, the resulting TLB way data is read out through the multiplexer. This data contains a previously cached value of the correct Page Table Entry (PTE) corresponding to the virtual address presented to the TLB, which holds the Real Page Number (RPN) and Page Attributes.

Finally, if this is an instruction address translation, the Page Attributes are checked to insure that the Page is not marked guarded (G = '1') or no execute (N = '1') (as well as all other architected checks to ensure access is allowed). If this condition is violated, an Instruction Storage Interrupt (ISI) is generated and the virtual translation step is complete. Otherwise, if this is a data transaction or if the conditions are not violated, then the RPN and Page Attributes are sent back to the PPE (to the ERAT), and the virtual translation step is complete.

Figure 8-2. Mapping a Virtual Address to a Real Address


80-bit Virtual Address (VA) (Bits [0:14] are zero in this implementation)
Virtual Page Number (VPN)
52
28–p
p
VSID
Page
Byte
0
51 52
79–p 80–p
79
TLB
PPE TLB Index Hash Function
Way A
Way B
Way C
Way D
Tlb_Tag
LPID[27:31]
PPE_TLB_Index[0:7]
Congruence Class
Valid
L II LP (from SLB Attributes)
Cmp
Cmp
Cmp
Cmp
Multiplexer
62–p
p
Byte
2
61–p 62–p
61
62-bit Real Address (RA) (Bits [2:21] are 0 in this implementation)

PPE TLB Index Hash Function:
- 4 KB: (VA[52:55] xor VA[60:63]) II VA[64:67]
- 64 KB: (VA[52:55] xor VA[56:59]) II VA[60:63]
- 1 MB: VA[52:59]
- 16 MB: VA[48:55]

Table 8-3. TLB Compare Tag

Page Size	Compare Tag
4 KB	VA[0:59] || Valid || L || LP || LPID[27:31]
64 KB	VA[0:55] || Valid || L || LP || LPID[27:31]
1 MB	VA[0:51] || Valid || L || LP || LPID[27:31]
16 MB	VA[0:47] || Valid || L || LP || LPID[27:31]

8.4.1. TLB Invalidation

Software must use the PowerPC Architecture **tlbie** or **tlbiel** instructions to invalidate entries in the TLB. The **tlbie** instruction is broadcast to all processors in the system. The architecture requires that only one processor per logical partition can issue a **tlbie** at a time.

This implementation supports bits [22:(63–p)], of the RB source register for **tlbie** and **tlbiel**, which results in a selective invalidation in the TLB based on VPN[38:(79–p)] and the page size. In other words any entry in the TLB with matching VPN[38:(79–p)] and page size is invalidated.

This processor implementation adds new fields to the RB register of the **tlbiel** instruction that are not currently defined in the architecture. These include the LP (large page selector) and IS (invalidation selector) fields.

The IS field is provided in RB[52:53] of the **tlbiel** instruction. *Table 8-4* gives details of the implementation of the IS field. Bit 1 of the IS field is ignored. For further details, see the Register Transfer Language (RTL) description given below.

Table 8-4. Summary of Supported IS Values in TLBIEL

IS	Behavior
00	The TLB is as selective as possible when invalidating TLB entries. The invalidation match criteria is VPN[38:79–p], L, LP, and LPID.
01	Reserved. Implemented the same as IS = '00'.
10	Reserved. Implemented the same as IS = '11'.
11	The TLB does a congruence class (index-based) invalidate. All entries in the TLB matching the index of the VPN supplied are invalidated.

The modified RTL definition for **tlbiel** specific to this implementation is as follows:

TLB Invalidate Entry Local X-form

tlbiel RB, L

31	///	L	///	RB	274	/
0	6	10	11	16	21	31

```
inval_sel ← $(RB)_{52}$
if inval_sel = 0 then
   if L = 0 then pg_size ← 4 KB
   else
      LP ← $(RB)_{51}$
      if LP = 0 then
            pg_size = large page size 1 selected by $HID6[LB]_{16:17}$
      else
            pg_size = large page size 2 selected by $HID6[LB]_{18:19}$
   p ← log_base_2(pg_size)
   for each TLB entry
      if ($entry\_VPN_{38:79-p}$ = $(RB)_{22:63-p}$) & (entry_pg_size = pg_size) then
            TLB entry ← invalid
else
   if entry_pg_size = 4 KB then
      if ($entry\_VPN_{52:55} \oplus entry\_VPN_{60:63}$ || $entry\_VPN_{64:67}$) = $(RB)_{44:51}$ then
            TLB entry ← invalid
   else if entry_pg_size = 64 KB then
      if ($entry\_VPN_{52:55} \oplus entry\_VPN_{56:59}$ || $entry\_VPN_{60:63}$) = $(RB)_{44:51}$ then
            TLB entry ← invalid
   else if entry_pg_size = 1 MB then
      if $entry\_VPN_{52:59}$ = $(RB)_{44:51}$ then
            TLB entry ← invalid
   else if entry_pg_size = 16 MB then
```

```
if entry_VPN$_{48:55}$ = (RB)$_{44:51}$ then
     TLB entry ← invalid
```

Let the invalidation selector be $(RB)_{52}$. If the invalidation selector is 0, then let all TLB entries that have the following properties be made invalid on the processor which executes this instruction.

- The entry translates a virtual address for which $VPN_{38:79-p}$ is equal to $(RB)_{22:63-p}$.
- The page size of the entry matches the page size specified by the L and LP field of the instruction.

If the invalidation selector is 1, then let all TLB entries that are in the congruence class of the TLB corresponding to $(RB)_{44:51}$ be made invalid on the processor which executes this instruction.

Since the ERAT breaks large pages into multiple 4-KB page entries, any **tlbie** or **tlbiel** instruction to a large page (L = '1') causes a complete invalidation of all entries in the ERAT. For a **tlbie** or **tlbiel** instruction to a small page (L = '0'), any I-ERAT or D-ERAT entries that have VPN[63:67] matching RB[47:51] of the instruction are invalidated.

Programming Note:

It is possible for a **tlbiel** with L = '0' and IS[0] = '1' to invalidate a TLB entry in a congruence class of the TLB but not invalidate the same translation in the I-ERAT or D-ERAT. It is software's responsibility in this case to make sure that ERAT coherency is maintained. Software can set L = '1' to avoid this (with the performance consequence of invalidating the entire ERAT).

The **tlbie** instruction RTL is similar except that IS[0] = '0' is assumed and the instruction is performed on all processors belonging to the same partition as the processor issuing the **tlbie** instruction.

Programming Note:

Since a **tlbiel** instruction to a large page causes the ERAT to be invalidated, software can issue a **tlbiel** to a large page that is not currently in use (unmapped in the page table) to cause an ERAT invalidation (RB = 0 is a convenient way to do this) without invalidating any entries in the TLB. This is useful for invalidating ERAT entries during a partition context switch. The IS field of **tlbiel** should be set to '00' for this operation.

The **tlbia** instruction is not supported by this implementation. To remove the entire contents of the TLB, the hypervisor should perform 256 **tlbiel** operations with IS set to '11' and RB[44:51] set to increment through each TLB congruence class.

8.4.2. Large Page-Size Decode

This implementation supports two large page sizes concurrently in the system (in addition to the small 4-KB page size). Each page can be a different size, and each segment must consist of pages that are all of the same size.

Two large page sizes can be chosen to run concurrently from a selection of three large pages (64 KB, 1 MB, and 16 MB). To accomplish this the implementation defines the LP field to override the low-order bit of the RPN in the PTE and the low-order bit of the VPN in the **tlbie**(**l**) instruction if the large page (L) bit is set to '1' for the PTE or **tlbie**(**l**) instruction. Whenever L = '1', RPN[51] and VPN[67] are implicitly zero. This is why the implementation can override these bits to be a large page selector (LP) whenever L = '1'.

There are two possible values of the LP field ('0' and '1'), which allows two different large page sizes to be concurrently active in the system. To determine the actual page size, four bits are kept in HID6[LB] (bits [16:19]). *Table 8-5* shows how the page size is determined given L, LP, and LB, where “Don’t Care” means that any value is allowed.

Programming Note:

Each partition in the system is allowed to have a different LB value. However, the partition is never allowed to change this value after the partition is created. In other words, the hypervisor is permitted to change the LB field in HID6 only if it is switching to or creating a partition. The partition is not allowed to dynamically change this value. The following sequence (or a similar one) must be issued in order to clean up translation after changing LB.

1. Issue **slbia** for thread 0
2. Issue **slbia** for thread 1
3. for (i = 0; i < 256; i++) **tlbiel** IS = 11 RB[44:51] = i; // Invalidate the entire TLB by congruence class
4. **sync** L = 0
5. **mtspr** HID6
6. **sync** L = 0
7. **isync**

Table 8-5. Large Page Decode

L	LP	LB[0:1]	LB[2:3]	Page Size
0	Don’t Care	Don’t Care	Don’t Care	4 KB
1	0	‘11’	Don’t Care	Reserved
1	0	‘10’	Don’t Care	64 KB
1	0	‘01’	Don’t Care	1 MB
1	0	‘00’	Don’t Care	16 MB
1	1	Don’t Care	‘11’	Reserved
1	1	Don’t Care	‘10’	64 KB
1	1	Don’t Care	‘01’	1 MB
1	1	Don’t Care	‘00’	16 MB

8.4.3. Tablewalk

When a virtual address is presented to the TLB and no matching TLB entry is found, the hardware initiates a reload of the TLB from the system Page Table if set to "hardware tablewalk" mode, which is indicated by LPCR[TL] = '0'. A "software tablewalk" reload mechanism can be used to reload the TLB instead by setting LPCR[TL] = '1'. If the LPCR[TL] bit is set to '1', then a page fault is generated immediately following any TLB miss. The MMU does not attempt to reload the TLB itself. For details on software accessibility and maintenance of the TLB, see *Section 12.3 TLB Management Instructions and ERAT Coherency* on page 125.

Programming Note:

- Each partition can set LPCR[TL] differently. Changes to its value must follow all the same architected synchronization rules as are required when changing the LPID.

The only occasion where the MMU performs a speculative tablewalk is a caching-inhibited store operation that takes a misaligned interrupt. Otherwise, the MMU waits to perform the tablewalk until the instruction is past the point at which any exception can occur.

The following notes apply to the hardware tablewalk search:

- The maximum Hash Table Size (HTABSIZE) allowed in Storage Description Register 1 (SDR1) is 24 ('11000') because the implementation only supports a 42-bit RA.
- The maximum Hash Table Origin (HTABORG) allowed in SDR1 is x'FFFFFF' because the implementation only supports a 42-bit RA.

PTEs are organized in memory in groups of eight entries called PTE Groups (PTEGs). A tablewalk consists of searching through a primary PTEG and then a secondary PTEG to find the correct PTE to be reloaded into the TLB.

The PTEG is searched in the following order:

1. Request Primary PTE
2. Search PTE[0], PTE[2], PTE[4], PTE[6]
3. Re-request Primary PTE
4. Search PTE[1], PTE[3], PTE[5], PTE[7]
5. Repeat steps 1 through 4 with the Secondary PTE
6. If no match occurs, raise a Data Storage exception

For best performance, the page table should be constructed to exploit this search order.

Table 8-6 summarizes the implemented bits of the PTE. Software should set all nonimplemented bits in the Page Table to zero. *Table 8-6* views the PTE as a quadword when giving bit definitions. For more information, see the *Page Table* section (*Section 4.5.1) of Book III: PowerPC Operating Environment Architecture.*

Table 8-6. PTE Implemented Bits

Bit(s)	Field	Description
0:14	Reserved	Software must set to 0 or results are undefined.
15:56	AVPN	Abbreviated Virtual Page Number
57:60	SW	Available for software use
61	L	Large Page Indicator
62	H	Hash function identifier
63	V	Valid
64:85	Reserved	Software must set to 0 or results are undefined.
86:**115**	RPN	Real Page Number
115	LP	Large Page Selector (last bit of RPN if L = '1')
116:117	Reserverd	Software must set to 0 or results are undefined.
118	AC	Address Compare bit
119	R	Reference bit
120	C	Change bit
121	W	Write-through. Hardware treats this as always 0. See *Section 8.4.3.1* on page 76.
122	I	Caching-inhibited bit
123	M	Memory-Coherence. Hardware treats this as always 1. See *Section 8.4.3.1* on page 76.
124	G	Guarded bit
125	N	No Execute bit
126:127	PP[0:1]	Page Protection bits 1:2

Page Table Match Criteria

The page table match criteria is described in the Page Table Search section of *Book III: PowerPC Operating Environment Architecture*. This processor implementation further requires that PTE[LP] = SLBE[LP] whenever PTE[L] = '1'. In other words, the page table entry page size must match the segment page size exactly for a page table match to occur. In addition, if more than one match is found in the page table, then the matching entries must all have the same PTE[LP] value or the results are undefined (see the Programming Note below). The PTE[LP] bit exists in the page table if and only if PTE[L] = '1'. When this is true, PTE[115] is no longer part of the PTE[RPN] field and instead serves as the PTE[LP] bit. This is done to conserve bits in the page table entry.

Programming Note:

- If multiple entries in the page table result in a match yet differ in any field other than SW, H, AC, R, or C, then the results are undefined. The implementation either returns the first matching entry encountered or it logically ORs the multiple matching entries together before storing them in the TLB. This can lead to spurious address translation results, and the system may checkstop as a result.
- This processor does not implement PTE[0:15]. Software must zero these bits in all page table entries. The processor ignores these bits if set, and a match occurs as if PTE[0:15] = x‘0000’.

8.4.3.1. WIMG Bits for Tablewalk

The processor always assumes that W = '0' and M = '1' regardless of how software may set W and M in the Page Table or TLB. All processor store operations are performed as non write-through and memory coherent (W = '0', M = '1') regardless of the state of these bits as set in the page table or as written by software to the TLB.

8.4.3.2. Reference and Change Bit Updates

If a hardware tablewalk is performed and a matching PTE entry is found with Reference bit set to '0', the MMU performs a PTE store update, with the Reference bit set to '1'. Therefore, since all tablewalks are nonspeculative (with the exception of a caching-inhibited misaligned store), the Reference bit always updates for any page that has a load or store request regardless of data or instruction storage interrupts that might occur.

If a hardware tablewalk is the result of a store operation, then a matching PTE is tested to see if the Change bit is set to '0'. If so, then the MMU performs a PTE store update, with the Change bit set to '1'. The only exception is a storage protection violation. The MMU does not update the Change bits if the storage access (load or store operation) results in a storage protection violation as described in the Storage Protection section (*Section 4.10) of Book III: PowerPC Operating Environment Architecture*. This processor performs all store operations in order.

Programming Note:

- A data storage operation to a page whose Change bit is zero in the TLB entry causes a page fault when in software tablewalk mode (LPCR[TL] = '1'). Software can set C = '1' on the initial load of a new TLB entry to avoid this if desired.

8.5. Real Addressing Mode

Table 8-7 summarizes how the real address is determined from the effective address in real addressing mode. (For "real addressing mode", see *Book III: PowerPC Operating Environment Architecture*).
A mode fault causes an instruction storage interrupt or a data storage interrupt (as appropriate).

Table 8-7. Summary of Real Addressing Modes

Mode Bits	Real Address Calculation	Descriptive Mode Name
MSR[HV] = ‘1’ EA(0) = ‘0’	RA = (EA[22:43] | HRMOR[22:43]) || EA[44:63]	Hypervisor Offset Mode
MSR[HV] = ‘1’ EA(0) = ‘1’	RA = EA[22:63]	Hypervisor Real Mode

MSR[HV] = ‘0’
LPES[1] = ‘1’
	RA = (EA[22:43] | RMOR[22:43]) || EA[44:63]	Real Offset Mode

MSR[HV] = ‘0’
LPES[1] = ‘0’
	LPAR Error (Interrupt)	Mode Fault

The following registers affect real-mode translations:

- RMOR
- HRMOR
- LPCR [RMLS, RMI, and LPES(1)]
- HID6[RMSC]

Real-mode translations are kept in the ERAT. Therefore, software must issue the following sequence when changing these registers:

- mtspr RMOR/HRMOR/LPCR[RMLS, RMI, LPES(1)]/HID6[RMSC]
- isync
- tlbiel L = ‘1’, IS = ‘00’, (RB) = 0
- sync L = 1

This form of **tlbie** causes all ERAT entries to be invalidated without any effect on the TLB.

Programming Note:

- Changing any of the above registers changes the context of all real-mode translations. Care must be taken to ensure that any previous real-mode translations are completed from a storage perspective and are invalidated from the ERAT. Software must not cause an implicit branch when changing any of these register values (as described in the implicit branch section of *Book III: PowerPC Operating Environment Architecture*).
- All real-mode accesses are treated as if they belong to a 4 KB page.

For details on how the I and G bits are set in real mode, see *Section 8.5.1 PPE Real-Mode Storage Control Facility* on page 79. Other attribute bits are set in real mode as shown in *Table 8-8*.

Table 8-8. Summary of Real Adderssing Mode Attributes

Attribute Bit	Value	Attribute Description
L	0	Large Page
LP	0	Large Page Selector
Ks	0	Supervisor State Storage Key
Kp	0	Problem State Protection Key
CL	0	Class

AC	0	Address Compare
PP[0:1]	10	Page Protection
C	1	Change
N	0	No Execute

8.5.1. PPE Real-Mode Storage Control Facility

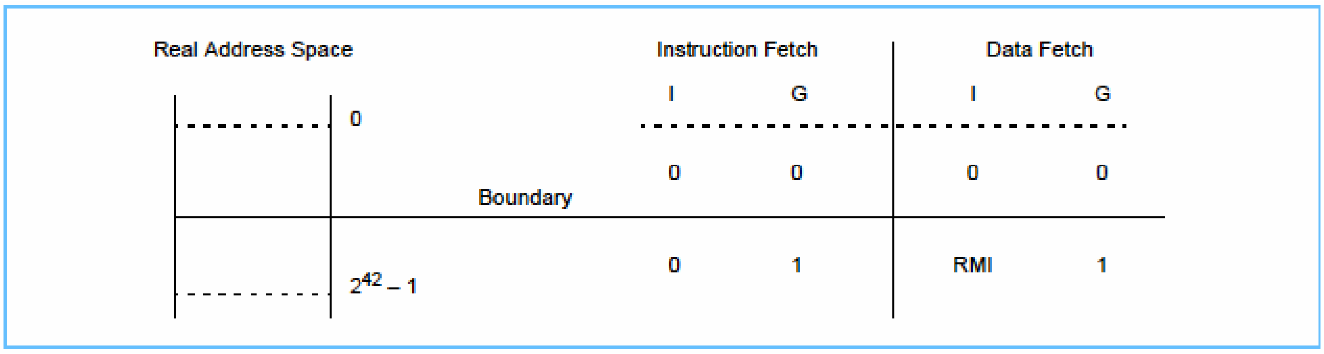
This implementation supports a real-mode storage control facility (see the related information in *Book III: PowerPC Operating Environment Architecture*). This facility allows for altering the storage control attributes in real mode based on a boundary in the real address space. The value in the 4-bit RMSC field stored in HID6[26:29] determines the boundary as shown in *Table 8-9*.

Table 8-9. Summary of Real-Mode Storage Control Values

RMSC(0:3)	Real Address Boundary
0000	0
0001	256 MB
0010	512 MB
0011	1 GB
0100	2 GB
0101	4 GB
0110	8 GB
0111	16 GB
1000	32 GB
1001	64 GB
1010	128 GB
1011	256 GB
1100	512 GB
1101	1 TB
1110	2 TB
1111	4 TB

Figure 8-3 demonstrates how the I (caching-inhibited) and G (guarded) real-mode attribute bits are set according to the location of the real address of the storage access relative to the boundary. RMI is the real-mode caching-inhibited bit (the RMI bit is located in LPCR[62]).

Figure 8-3. Real-Mode Storage Control I and G Bit Settings



9. Exceptions and Interrupts

This section describes the exceptions outlined in the PowerPC Architecture and the implementation specific details of the PowerPC Processor Element-specific exceptions. Most exceptions are defined in the PowerPC Architecture. However, some exceptions have implementation-specific details.

Please refer to *Chapter 5 Interrupts of Book III: PowerPC Operating Environment Architecture, Version 2.02* to understand the general idea of the interrupt mechanism of the PowerPC Architecture.
The PowerPC Processor Element also has the hypervisor state to support the Logical Partitioning facility. Please refer to *Section 1.7 Logical Partitioning(LPAR) of Book III: PowerPC Operating Environment Architecture, Version 2.02* for more details on this function.

Book III: PowerPC Operating Environment Architecture, Version 2.02 defines two types of exception conditions:

1. Exceptions caused directly by the execution of an instruction (instruction-caused exceptions)
2. Exceptions caused by an asynchronous external event (system-caused exceptions)

The occurrence of an exception produces an interrupt that, if enabled, causes the processor to execute an interrupt handler.

Interrupts are the mechanism by which a processor identifies the exception condition, generates the interrupt, and passes control to the interrupt handler.

Exceptions are the condition that causes the processor interrupt, if the interrupt is enabled. It is possible for a single instruction to cause multiple exceptions and for multiple exception conditions to cause a single interrupt. In the latter case, additional status is provided for software to determine the cause of the interrupt. Interrupts are also produced in response to external events, such as the assertion of an External interrupt due to a device requiring service, or other system caused interrupts, such as a Decrementer, Hypervisor Decrementer, or Thermal interrupt. In these cases, the status for the cause of the interrupt is usually provided by the external device or the system interrupt controller.

9.1. Exception Classification

As specified by the PowerPC Architecture, exceptions are either system-caused or instruction-caused. Instruction-caused exceptions are further refined as precise or imprecise. Depending on the condition of the processor when the interrupt is taken, certain interrupts are either recoverable or nonrecoverable. All Instruction-caused exceptions in the PPE are precise (with the exception of the Machine Check interrupt; see the note below). Precise means the exception is generated when the instruction is fetched or executed. *Book III: PowerPC Operating Environment Architecture, Version 2.02* provides the definition of a precise interrupt. System-caused interrupts are caused by external events and are asynchronous to processor execution. The types of exceptions supported by the PPE are listed in *Table 9-1* on page 81.

Note:

- Although the PowerPC Architecture defines the machine check interrupt as a system-caused interrupt, this implementation treats the machine check interrupt as an instruction-caused interrupt. It is considered an instruction-caused interrupt because the only source of the machine check interrupt is a data error on a caching inhibited load instruction. In the sections that follow, the machine check interrupt is regarded as instruction-caused (from the point of view of the thread issuing the offending load instruction). Therefore, it is designated as both precise and imprecise.
- The PowerPC Architecture defines imprecise floating-point exception modes. The PPE treats all floating-point exception modes as precise.

Table 9-1. PPE Exception Classifications

Interrupt Classes	Precise/Imprecise	Exception Types
System-caused, Nonmaskable	N/A	The System Reset interrupts are asynchronous to the instruction stream execution.
System-caused, Maskable	N/A	The System Error, External, Decrementer, Hypervisor Decrementer, Maintenance, and Thermal Management interrupts are asynchronous to the instruction stream execution and can be masked by the processor state.
Instruction-caused	Precise	Exceptions that arise during the fetch, execution, or the completion of an instruction.
	Imprecise	Exceptions that arise during the completion of an instruction. The only interrupt in this class is the Imprecise Machine Check interrupt.

9.2. Exception Handling

Exceptions are handled by an interrupt facility, defined in *Book III: PowerPC Operating Environment Architecture, Version 2.02* that allows the processor to change to a privileged state in response to external events, errors, or exception conditions during the execution of an instruction.

Upon entering the privileged state, a small portion of the processor's current state is saved in the Machine Status Save/Restore Registers (the SRR0, SRR1, HSRR0, and HSRR1 registers), the Machine State Register (MSR) is updated, and the instruction fetch and execution resumes at the real address associated with the interrupt (the interrupt vector). Since the Machine Status Save/Restore registers are serially reused by the processor, all interrupts with the exception of System Reset and Machine Check are ordered as defined in the PowerPC Architecture.

Either the SRR0 or the HSRR0 register is set to the effective address of the instruction where processing should resume when returning from the exception handler. Depending on the interrupt type, the effective address might be the address of the instruction that caused the exception or the next instruction the processor would have executed if an interrupt condition did not exist. With the exception of an Imprecise Machine Check interrupt, all interrupts are context-synchronizing as defined in *Book III: PowerPC Operating Environment Architecture, Version 2.02*. Essentially, all instructions in the program flow preceding the instruction pointed to by SRR0 or HSRR0 have completed execution and no subsequent instruction has begun execution when the interrupt is taken. The program restarts from the address of SRR0 or HSRR0 when returning from an interrupt (the execution of an **rfid** or **hrfid** instruction).

Note: In this document, the phrase *the interrupt is taken* refers to the PowerPC Architecture interrupt facility.

Because the MSR setting is modified when the interrupt is taken, the SRR1 or the HSRR1 register is used to save the current MSR state and information pertaining to the interrupt. Bits [33:36] and [42:47] of the SRR1 or HSRR1 registers are loaded with information specific to the interrupt type. The remaining bits in the SRR1 or the HSRR1 register (bits [0:32], [37:41], and [48:63]) are loaded with a copy of the corresponding bits in the MSR. The MSR bits saved in the SRR1 or HSRR1 registers are restored when returning from an interrupt (the execution of a **hrfid** or **rfid** instruction).

The PPE supports two threads of execution. Each thread is viewed as an independent processor complete with separate exceptions and interrupt handling. Exceptions can occur simultaneously for each thread. The PPE supports concurrent handling of exceptions on both threads by duplicating some registers defined by the PowerPC Architecture.

The following registers associated with exception handling are duplicated or are thread dependent:

- Machine State Register (MSR)
- Machine Status Save/Restore Registers (SRR0 and SRR1)
- Hypervisor Machine Status Save/Restore Registers (HSRR0 and HSRR1)
- Floating-Point Status and Control Register (FPSCR)
- Data Storage Interrupt Status Register (DSISR)

- Decrementer (DEC)
- Logical Partition Control Register (LPCR)
  Note: The LPCR is partially shared register. The LPCR[RMLS], LPCR[TL], and LPCR[LPES] fields are shared between threads; the LPCR[MER], LPCR[RMI], and LPCR[HDICE] fields are duplicated per thread.
- Data Address Register (DAR)
- Data Address Breakpoint Register (DABR and DABRX)
- Address Compare Control Register (ACCR)
- Thread Status Register Local (TSRL)
- Thread Status Register Remote (TSRR)

In addition, the following thread-independent registers also are associated with exception handling on both threads:

- Hypervisor Decrementer (HDEC)
- Control Register (CTRL)
- Hardware Implementation Dependent Register 0 (HID0)
- Hardware Implementation Dependent Register 1 (HID1)
- Thread Switch Control Register (TSCR)
- Thread Switch Time-Out Register (TTR)

9.3. Interrupt Priorities and Definitions

9.3.1. Interrupt Priorities

The following exception conditions are ordered from highest to lowest priority.

I. System Reset Interrupt (highest priority exception)
II. Machine Check Interrupt
III. Instruction Dependent
  A. Fixed-Point Loads and Stores
  B. Floating-Point Loads and Stores
  C. Other Floating-Point Instructions
  D. VXU Loads and Stores
  E. Other VXU Instructions
  F. **rfid**, **hrfid**, and **mtmsr**[**d**]
  G. Other Instructions
  H. Instruction Segment Interrupt
  I. Instruction Storage Interrupt
IV. Thermal Management Interrupt
V. System Error Interrupt
VI. Maintenance Interrupt
VII. External Interrupt
  A. Direct
  B. Mediated
VIII. Hypervisor Decrementer Interrupt
XI. Decrementer Interrupt

9.3.2. Interrupt Definitions

Table 9-2 shows the types of interrupts that can occur and the settings of the MSR bits and alteration of related registers relative to the interrupt type.

Table 9-3 on page 84 list the interrupts supported by the PPE, the effective address of the interrupt handler (interrupt vector).

Note: The PPE does not implement the optional Performance Monitor interrupt defined in the PowerPC Architecture. In addition, the PPE does not implement the example extensions (optional) to the trace facility as outlined in an appendix to the *Book III: PowerPC Operating Environment Architecture, Version 2.02*.

Table 9-2. Registers Altered by Interrupts

Interrupt Type	MSR Bit[1]			SRR0,SRR1 / HSRR0,HSRR1	Other Registers
	HV	ME	RI[2]		
System Reset	1	—	0	S [5]	TSRL[11:12] [6], CTRL[8:9] [7]
Imprecise Machine Check	1	0	0[3]	S	
Precise Machine Check	1	0	0[3]	S	
Data Storage	m	—	0	S	DSISR, DAR
Data Segment	m	—	0	S	DSISR, DAR
Instruction Storage	m	—	0	S	
Instruction Segment	m	—	0	S	
External	e	—	— / 0[4]	S/H	
Alignment	m	—	0	S	DAR[8]
Program	m	—	0	S	
Floating-point unavailable	m	—	0	S	
Decrementer	m	—	0	S	
Hypervisor Decrementer	1	—	—	H	
System Call	s	—	0	S	
Trace	m	—	0	S	
VXU Unavailable	m	—	0	S	
System Error	1	—	—	H	
Maintenance	1	—	—	H	
Thermal Management	1	—	—	H	

Legend:

0	Bit is set to '0'.
1	Bit is set to '1'.
—	Bit is not altered
m	MSR[HV] is set to '1' if LPCR[LPES0] = '0' and LPCR[LPES1] = '0'; otherwise the state of MSR[HV] is not altered.
e	MSR[HV] is set to '1' if LPCR[LPES0] = '0'; otherwise the state of MSR[HV] is not altered.
s	MSR[HV] is set to '1' if LEV = 1 or (LPCR[LPES0] = '0' and LPCR[LPES1] = '0'); otherwise the state of MSR[HV] is not altered.
S	SRR0 and SRR1 are updated.
H	HSRR0 and HSRR1 are updated.
S/H	SRR0 and SRR1 are updated if LPCR[LPES0] = ‘1’ or HID0[extr_hsrr] = ‘0’. HSRR0 and HSRR1 are updated if LPCR[LPES0] = ‘0’ and HID0[extr_hsrr] = ‘1’.

Note:
1. MSR[BE], MSR[FP], MSR[PR], MSR[SE], and MSR[IR] are set to '0', and MSR[SF] is set to '1'.
2. MSR[RI] is not changed for interrupts that use HSRR0 or HSRR1.
3. MSR[RI] is set to '0'. SRR1[62] is set to '0' for Imprecise Machine Check and is not altered for Precise Machine Check.
4. MSR[RI] is not altered if LPCR[LPES0] is set to '0' and HID0[extr_hsrr] is set to'1'; otherwise, MSR[RI] is set to '0'.
5. SRR1[42:44] includes System reset reason code.
   - 000 Reserved
   - 001 Reserved
   - 010 System reset due to a thermal management interrupt to a suspended thread
   - 011 System reset due to a decrementer interrupt to a suspended thread
   - 100 System reset due to an external interrupt to a suspended thread
   - 101 Thread resumed due to one of the following conditions:
     - A System Reset due to POR
     - A write to CTRL with the thread enable bit set
   - 110 Thread resumed due to a System Error
   - 111 Reserved
6. TSRL[11:12] = '11' for High thread priority.

7. CTRL[8:9]: Depending on which thread took the interrupt, the corresponding thread enable bit (bit 8 for thread0, bit9 is for thread1) is set.
8. The PPE does not modify the DSISR when an alignment interrupt is taken.

Table 9-3. Interrupt Vector

Exception Type	Vector Offset	
	Thread 0	Thread 1
System Reset	Selectable	x’00..00000100’
Machine Check	x’00..00000200’	x’00..00000200’
Data Storage Int.	x’00..00000300’	x’00..00000300’
Data Segment Int.	x’00..00000380’	x’00..00000380’
Instruction Storage Int.	x’00..00000400’	x’00..00000400’
Instruction Segment Int.	x’00..00000480’	x’00..00000480’
External	x’00..00000500’	x’00..00000500’
Alignment	x’00..00000600’	x’00..00000600’
Program	x’00..00000700’	x’00..00000700’
Floating Point Unavailable	x’00..00000800’	x’00..00000800’
Decrementor	x’00..00000900’	x’00..00000900’
Hypervisor Decrementor	x’00..00000980’	x’00..00000980’
System Call	x’00..00000C00’	x’00..00000C00’
Trace	x’00..00000D00’	x’00..00000D00’
VXU Unavailable	x’00..00000F20’	x’00..00000F20’
System Error	x’00..00001200’	x’00..00001200’
Maintenance	x’00..00001600’	x’00..00001600’
Thermal Management Int.	x’00..00001800’	x’00..00001800’

9.3.3. Mediated External Interrupt

The PPE implements a mediated external interrupt extension to the PowerPC Architecture for external interrupts.
Section 9.3.3.1 defines the extension; *Section 9.3.3.2* on page 86 provides implementation details.

9.3.3.1. Mediated External Interrupt Architecture

The new kind of external exception is called a mediated external exception.
The currently defined external exception is called a direct external exception.
An external interrupt that is caused by a mediated external exception is called a mediated external interrupt.
Correspondingly, an external interrupt that is caused by a direct external exception is called a direct external interrupt.
Bit 52 of the LPCR is defined as the Mediated External Exception Request (MER) bit.
The value ‘1’ means that a mediated external exception is requested. The value ‘0’ means that a mediated external exception is not requested.

On a shared processor, the hypervisor sets bit zero of the logical partitioning environment selector to zero (LPES[0] = ‘0’) to cause external interrupts to go to the hypervisor.
In the current architecture, external interrupts are disabled if MSR[EE] = ‘0’, and MSR[EE] can be altered by the operating system. Thus, by running with MSR[EE] = ‘0’, an operating system can delay the presentation of external interrupts to the hypervisor for nontrivial periods of time.

The mediated external interrupt extension allows hypervisor software to service external interrupts even when external interrupts are disabled (MSR[EE] = ‘0’). This feature reduces the external interrupt latency for applications running in a partitioned system.

On a shared processor with LPES[0] set to ‘0’, when redispatching a partition if an external interrupt has occurred for the partition and has not yet been presented to the partition, the hypervisor passes control to the operating system’s external interrupt handler as described below.

- If the external interrupt was Direct and the partition has MSR[EE] = ‘1’, the hypervisor performs these steps:
  - Sets registers (MSR and SRR0/1, and the external interrupt hardware registers as appropriate) to emulate the external interrupt.
  - Returns to the operating system’s external interrupt handler.
  - Restores the partition's LPCR[MER].
- If the external interrupt was Direct but the partition has MSR[EE] = ‘0’, the hypervisor performs these steps:
  - Returns to the partition at the instruction at which it was interrupted.
  - Sets LPCR[MER] to ‘1’.
- If the external interrupt was Mediated and the partition now has MSR[EE] = ‘1’, the hypervisor performs these steps:
  - Sets registers (MSR and SRR0/1, and external interrupt hardware registers as appropriate) to emulate the original direct external interrupt.
  - Returns to the operating system’s external interrupt handler.
  - Sets LPCR[MER] to ‘0’ if all external interrupts (for the partition) now have been presented to the partition. Otherwise, restore the partition's LPCR[MER].

In all three cases, the partition is redispatched with MSR[EE] = ‘0’. When the partition is to be redispatched at the operating system’s external interrupt handler (as in the first and third cases), the hypervisor sets the MSR and SRR0/1 as if the original direct external interrupt occurred when LPES[0] was set to ‘1’ and the partition was executing. In particular, no indication is provided to the operating system (for example, in an SRR1 bit) regarding whether the external interrupt that is now being presented to the partition was Direct (first case) or Mediated (third case).

The MER bit has the same relationship to the existence of a mediated external exception as bit zero of the Decrementer (DEC[0]) or Hypervisor Decrementer (HDEC[0]) has to the existence of a decrementer or hypervisor decrementer exception. (See PowerPC Architecture, Book III for more information.)
The exception effects of LPCR[MER] are considered consistent with the contents of LPCR[MER] if one of the following statements is true.

- LPCR[MER] = ‘1’ and a mediated external exception exists.
- LPCR[MER] = ‘0’ and a mediated external exception does not exist.

A context-synchronizing instruction or event that is executed or occurs when LPCR[MER] equals ‘0’ ensures that the exception effects of LPCR[MER] are consistent with the contents of LPCR[MER].
Otherwise, when an instruction changes the contents of LPCR[MER], the exception effects of LPCR[MER] become consistent with the new contents of LPCR[MER] reasonably soon after the change.

9.3.3.2. Mediated External Interrupt Implementation

As described in *Section 9.3.3.1*, this extension defines two conditions that cause an external interrupt:

- The assertion of an external interrupt signal causes a direct external interrupt.
  It is taken if the following expression is equal to ‘1’:
  ```
  MSR[EE] | (^LPCR[LPES0] & (^MSR[HV] | MSR[PR]) & HID0[extr_hsrr])
  ```
- The hypervisor setting a mediated external interrupt request (by setting LPCR[MER] = ‘1’) causes a mediated external interrupt. It is taken if the following expression is equal to ‘1’:
  ```
  MSR[EE] & (^MSR[HV] | MSR[PR]) & HID0[extr_hsrr]
  ```

For direct external interrupts, the external interrupt signals must remain asserted until the interrupt is taken.
Deasserting an external interrupt signal before the PPE takes the interrupt can lead to undefined results.
After the PPE begins execution of the external interrupt handler, the external interrupt signal can be deasserted.

The PPE treats the external interrupt conditions as a context-synchronizing operation as defined in the PowerPC Architecture. When the PPE takes an external interrupt, the register state is altered as defined in *Table 9-2* on page 83. Instruction fetch and execution resume at the effective address specified in *Table 9-3* on page 84.

9.3.4. Miscellaneous PPE-Specific Interrupts

The PPE implements several PPE-specific interrupts not defined in the PowerPC Architecture.
The PPE treats these interrupts as context-synchronizing operations as defined in the PowerPC Architecture.
When the PPE takes one of these interrupts, the register state is altered as defined in *Table 9-2* on page 83.
Instruction fetch and execution resume at the effective address specified in *Table 9-3* on page 84.

9.3.4.1. VXU Unavailable Interrupt

A VXU unavailable interrupt occurs when a higher-priority interrupt does not exist, an attempt is made to execute a VXU instruction, and MSR[VXU] is set to ‘0’.
VXU unavailable interrupts are instruction-caused, precise, and cannot be masked.

9.3.4.2. System Error Interrupt

A system error interrupt occurs when no higher-priority interrupts exists, a system error interrupt signal is asserted for one of the execution threads, and the system error interrupts are enabled.
System error interrupts are system-caused and are enabled if the following expression is equal to ‘1’:

```
(MSR[EE] | ^MSR[HV]) & HID0[en_syserr]
```

9.3.4.3. Maintenance Interrupt

The maintenance interrupt is intended for hardware debugging purposes only. This interrupt is not intended for normal programming use.
System-caused maintenance interrupts are enabled if the following expression is equal to '1':

```
    MSR[EE] | ^MSR[HV]
```

Instruction-caused maintenance interrupts cannot be masked.

9.3.4.4. Thermal Management Interrupt

A thermal management interrupt occurs when no higher-priority interrupts exist, a thermal management interrupt signal is asserted for one of the two execution threads, and the thermal management interrupts are enabled. Thermal management interrupts are system-caused and can be masked.
Thermal management interrupts are enabled if the following expression is equal to '1':

```
    (MSR[EE] | ^MSR[HV]) & HID0[therm_intr_en]
```

10. Performance

This section provides information about many of the key latencies, throughput rates and bandwidths in the PowerPC Processor Element.

10.1. Instruction Characteristics

The following table summarizes the PowerPC Processor Element raw instruction-level performance characteristics. Some comments on the table:

- An instruction is executed in one or more pipelines. The pipelines an instruction is dispatched to are noted in the table below.
- Latency is the issue-to-issue latency for a dependent instruction. For example, if an **add** issues in cycle 20 and the soonest a dependent **xor** could issue is cycle 22, the latency for **add** is listed as 2. There are multiple latencies for some instructions when the instruction writes more than one type of register. In general, latency is one cycle larger than use penalty.
- Throughput refers to the maximum sustained rate the PowerPC Processor Element can execute instructions of the type noted in the absence of any dependencies assuming infinite caches. It is shown as instructions/cycle (ipc).
- Some fields are labeled 'N/A' to indicate that a more elaborate description is required (and is best understood by taking a broader view of the machine as a whole). This includes complex microcoded instructions. Instructions that don't modify a register are also labeled 'N/A'.
- In the notes column, “MC” indicates instructions are microcoded. “CSI” indicates a context synchronizing instruction.

Table 10-1. Instruction Characteristics

Instruction	Execute			Notes	Comments
	Pipeline	Latency	Through put (ipc)		
b ba bl bla	BRU	N/A	1/cycle		Unconditional branches always "predicted" correctly.
bc bca bcl bcla	BRU	1 cycle (lr / ctr)	1/cycle		Branch direction predicted at top of pipeline. May update the link stack.
bclr bclrl	BRU	1 cycle (lr / ctr)	1/cycle		Branch direction and target address predicted at top of pipeline. May use the link stack.
bcctr bcctrl	BRU	1 cycle (lr) 2 cyc (ctr)	1/cycle		Branch direction and target address predicted at top of pipeline. May use the link stack.
sc rfid	IU	N/A	N/A	CSI	
crand cror crxor crnand crnor crandc creqv crorc	BRU	1 cycles	1/cycle		
mcrf	BRU	1 cycles	1/cycle		
lbz lbzx lhz lhzx lwz lwzx ld ldx lhbrx lwbrx	LSU	2 cycles	1/cycle		5-cycle latency for **load** followed by dependant **load**
lha lhax lwa lwax	LSU, FXU	N/A		MC	
lbzu lhzu lwzu ldu	LSU, FXU	2 cyc (RT) 2 cyc (RA)	1/cycle		Hardware breaks into basic **load** and an **add**
lbzux lhzux lwzux ldux	LSU, FXU	2 cyc (RT) 2 cyc (RA)	1/cycle		Hardware breaks into a basic **load** and an **add**
lhau	LSU, FXU	N/A		MC	
lhaux lxaux	LSU, FXU	N/A		MC	
stb sth stw std	LSU, FXU	N/A	1/cycle		
stbx sthx stwx stdx	LSU, FXU	N/A	1/cycle		

Table 10-1. Instruction Characteristics

Instruction	Execute			Notes	Comments
	Pipeline	Latency	Through put (ipc)		
stbu sthu stwu stdu	LSU, FXU	2 cycles for updated register	1/cycle		Hardware breaks into basic **store** and an **add**
sthbrx stwbrx	LSU, FXU	N/A	1/cycle		
stbux sthux stwux stdux	LSU, FXU	2 cycles for updated register	1/cycle		Hardware breaks into basic **store** and an **add**
lmw	LSU	N/A	1 register loads per cycle after startup	MC	Microcode engine generates inline sequence of basic loads
stmw	LSU, FXU	N/A	1 register stores per cycle after startup	MC	Microcode engine generates inline sequence of basic stores
lswi (naturally aligned)	LSU	N/A	1 register load per cycle after startup	MC	Microcode engine assumes natural alignment and generates inline sequence of basic loads
lswx (naturally aligned)	LSU	N/A	1 register load per cycle after startup	MC	Microcode engine assumes natural alignment and generates inline sequence of basic loads
stswi (naturally aligned)	LSU, FXU	N/A	1 register store per cycle after startup	MC	Microcode engine assumes natural alignment and generates inline sequence of basic stores
stswx (naturally aligned)	LSU, FXU	N/A	1 register store per cycle after startup	MC	Microcode engine assumes natural alignment and generates inline sequence of basic stores
lswi lswx stswi stswx (unaligned)	LSU	N/A		MC	A string instruction is first decoded and issued as described above. At execute, the LSU notes that it is unaligned and causes a machine flush. As the string instruction goes through microcode engine the second time, it is broken up in a way that takes the misalignment into account.
lwarx ldarx	LSU	N/A	N/A		Forced to miss data L1 cache One outstanding **lwarx** in system at a time.
stwcx. stdcx.	LSU, FXU	N/A	N/A		Must establish coherency block ownership before completing the instruction (other stores don't have to do this). Can take anywhere from 10 cycles to 100's depending upon the state of the coherency block in the memory hierarchy)
addi addis add subf neg	FXU	2 cyc (gpr)	1/cycle		
add. subf. subfic neg. subfe	FXU	2 cyc (gpr) 1 cyc (cr)	1/cycle		
addco subfco addeo subfeo addmeo subfmeo addzeo subfzeo nego addo addze subfo addc subfc addic subic adde addme subfme subfze	FXU	2 cyc (gpr) 2 cyc (xer)	1/cycle		
addic. adde. addze. subfe. addme. subfme. subfze. addo. subfo. subfeo. addeo. addmeo. subfmeo. addzeo. subfzeo. addc. addco. subfc. subfco.	FXU	2 cyc (gpr) 1 cyc (cr) 2 cyc (xer)	1/cycle		
mulli	FXU	6 cycles	1/6 cycle		Not pipelined in the FXU. Causes a 6-cycle stall after issue.

Table 10-1. Instruction Characteristics

Instruction	Execute			Notes	Comments
	Pipeline	Latency	Through put (ipc)		
mullw mulhw mulhwu mullwo	FXU	9 cycles	1/9 cycles		Not pipelined in the FXU. Causes a 9-cycle stall after issue.
mulld mulhd mulhdu mulldo	FXU	15 cycles	1/15 cycles		Not pipelined in the FXU. Causes a 15-cycle stall after issue.
mullw. mulld. mulhd. mulhw. mulhdu. mulhwu.	FXU			MC	Not pipelined in the FXU. Microcode engine breaks into baseline operation and a compare.
mullwo. mulldo.	FXU			MC	Not pipelined in the FXU. Microcode engine breaks into baseline operation and a compare.
divd divdu divdo divduo	FXU (one)	10 – 70 cycles	1/10 – 1/70 cycles		Not pipelined in the FXU. The XU uses complex queue, pipeline stalls until writeback. The fixed-point divide is a variable latency operation that calculates RA and RB for word or doubleword and signed or unsigned integer operands. Division is defined by the following equation: dividend = (quotient x divisor) + r where 0 <= r < |divisor| , when dividend >= 0 and –|divisor| < r <= 0, when dividend < 0 Overflow is set when an attempt is made to perform negativemax / (–1) or anything / 0. The performance is determined by the number of bits required to represent the result. PPU cycles equal: ((1 setup) + (ceil ((rb leading digits – ra leading digits) / 2) + 1 iterations) + (1 fixup)) × 2 word min = 10, max = 38 cycles doubleword min = 10, max = 70 cycles Overflow cases complete in 10 cycles
divw divwu divwo divwuo	FXU (one)	10 – 38 cycles	1/10 – 1/38 cycles		Not pipelined in the FXU. The XU uses complex queue, pipeline stalls until writeback. See the performance notes for **divd**
divd. divw. divdu. divwu.	FXU (one)			MC	Not pipelined in the FXU. Microcode engine breaks into a divide and a compare. The XU uses complex queue, pipeline stalls until writeback. See the performance notes for **divd**
divdo. divwo. divduo. divwuo.	FXU (one)			MC	Not pipelined in the FXU. Microcode engine breaks into a divide and a compare. See the performance notes for **divd**
cmpi cmp cmpli cmpl	FXU	1 cycle	1/cycle		
tdi twi td tw	FXU	N/A	1/cycle		
ori oris xori xoris and or xor nand nor eqv andc orc	FXU	2 cyc (gpr)	1/cycle		
andi. andis. and. or. xor. nand. nor. eqv. andc. orc. nego.	FXU			MC	
extsb extsh extsw	FXU	2 cyc	1/cycle		
extsb. extsh. extsw.	FXU			MC	
cntlzd cntlzw	FXU	2 cyc (gpr)	1/cycle		
cntlzd. cntlzw.	FXU			MC	

Table 10-1. Instruction Characteristics

Instruction	Execute			Notes	Comments
	Pipeline	Latency	Through put (ipc)		
rldicl rldic rldicr rlwinm rldimi rlwimi	FXU	2 cyc (gpr)	1/cycle		
rldicl. rldicr. rldic. rlwinm. rldcl rldcl. rldcr rldcr. rlwnm rlwnm. rldimi. rlwimi.	FXU			MC	
sld sld. slw slw. srd srd. srw srw. srad srad. sraw sraw. sradi. srawi.	FXU			MC	
sradi srawi	FXU	2 cyc (gpr)	1/cycle		
mtspr(xer)	FXU (one)	N/A	N/A		
mtspr(lr) mtspr(ctr)	BRU	1 cycle	1/cycle		
mtspr(others) mtmsr mtmsrd	either FXU or BRU (depends on SPR)	varies based on SPR	varies based on SPR		**mtmsrd** L = 0 is microcoded. **mtmsrd** L = 1 is not. LR and CTR have 1-cycle latency and issue 1/cycle.
mtcrf	FXU	N/A	N/A	MC	Bit 11='0'
mtocrf	FXU	1 cycle	1 cycle		Bit 11='1'
mcrxr	BRU	1 cycle	1/cycle		
mfcr	BRU	~34 cycles	~1/34cycles		Bit 11='0'. This form is nonpipelined.
mfocrf	BRU	~34 cycles	~1/34cycles		Bit 11='1'. This form is nonpipelined.
mftb	FXU (one)	~40 cycles	~2/40 cycles		
mfspr(lr) mfspr(ctr)	BRU	1 cycle	1/cycle		
mfspr(others) mfmsr	either FXU or BRU (depends on SPR)	varies based on SPR	varies based on SPR		Other SPRs are handled by the FXU and are nonpipelined. The pipeline stalls until writeback.
lfs lfsx lfd lfdx	LSU, FPU Load	1 cycle	1/cycle		Instructions are issued to both the FPU and the LSU units; only executes in the LSU unit
lfsu lfsux lfdu lfdux	LSU, FXU, FPU Load	1 cyc (fpr) 2 cyc (gpr)	1/cycle		
stfs stfsx stfd stfdx	LSU, FPU Store	N/A	1/cycle		Instructions are dispatched to both the FPU and the LSU units
stfsu stfsux stfdu stfdux	LSU, FPU, FXU	2 cycle for gpr	1/cycle		
stfiwx	LSU, FPU	N/A	1/cycle		
fmr fneg fabs fnabs fadd fadds fsub fsubs fmul fmuls	FPU	10 cycles	1/cycle		
fmr. fneg. fabs. fnabs. fadd. fadds. fsub. fsubs. fmul. fmuls.	FPU	10 cyc (fpr) +1 cyc(cr)	1/cycle		The IU creates stall condition for CR1 dependencies.
fmadd fmadds fmsub fmsubs fnmadd fnmadds fnmsub fnmsubs	FPU	10 cycles	1/cycle		
fmadd. fmadds. fmsub. fmsubs. fnmadd. fnmadds. fnmsub. fnmsubs.	FPU	10 cyc (fpr) +1 cyc (cr)	1 cycle		The IU creates stall condition for CR1 dependencies.
fdiv fdivs (IEEE)	FPU	74 cycles	1/74 cycles		Nonpipelined in the FPU.

Table 10-1. Instruction Characteristics

Instruction	Execute			Notes	Comments
	Pipeline	Latency	Through put (ipc)		
fdiv fdivs (non-IEEE)	FPU	56 cycles	1/56 cycles		Non-IEEE results are accurate to within 1 or 2 ULPs of the correctly rounded IEEE result, depending on rounding mode: 1 ULP for to-nearest and to-zero rounding modes, 2 ULP for the other modes. Nonpipelined in the FPU.
fsqrt fsqrts (IEEE)	FPU	84 cycles	1/84 cycles		Nonpipelined in the FPU.
fsqrt fsqrts (non-IEEE)	FPU	66 cycles	1/66 cycles		Non-IEEE results are accurate to within 1 or 2 ULPs of the correctly rounded IEEE result, depending on rounding mode: 1 ULP for to-nearest and to-zero rounding modes, 2 ULP for the other modes. Nonpipelined in the FPU.
fres	FPU	10 cycles	1/cycle		
frsqrte	FPU	10 cycles	1/cycle		
fdiv. fdivs. fsqrt. fsqrts. fres. frsqrte.	FPU	same as above +1 cyc (cr)	same as above		The IU creates stall condition for CR1 dependencies.
frsp	FPU	10 cycles	1/cycle		
fctid fctidz fctiw fctiwz fcfid	FPU	10 cycles	1/cycle		
frsp. fctid. fctidz. fctiw. fctiwz. fcfid.	FPU	same as above +1 cyc (cr)	1/cycle		The IU creates stall condition for CR1 dependencies.
fcmpu fcmpo	FPU	1 cycle	1/cycle		
fsel	FPU	10 cycles	1/cycle		
fsel.	FPU	same as above +1 cyc (cr)	1/cycle		
mffs mffs.	FPU	11-28 cycles	1/28 cycles		These instructions are stalled at VQ8 until all older VSU operations are complete.
mcrfs	FPU	14	N/A		MC
mtfsfi mtfsfi. mtfsf mtfsf. mtfsb0 mtfsb0. mtfsb1 mtfsb1.	FPU	1 cycle	1/cycle		
sync	LSU	N/A	N/A		The IU holds at issue until all queues and pipelines have drained for that thread. After issue the **sync** forces previous stores to finish into the cache or memory hierarchy; that is, out of the STQs.
lwsync	LSU	N/A	N/A		The IU holds at issue until all queues and pipelines have drained for that thread. After issue the **lsync** forces previous stores to finish into the cache or memory hierarchy; that is, out of the STQs. Still broadcasts **sync** transaction onto EIB (but doesn't block)
ptesync	LSU	N/A	N/A		The IU holds at issue until all queues and pipelines have drained for that thread. After issue the **ptesync** forces previous stores to finish into the cache or memory hierarchy; that is, out of the STQs. Still broadcasts **sync** transaction onto EIB (but doesn't block)
eieio	LSU	N/A	N/A		
isync	LSU	N/A	N/A	CSI	The IU holds at issue until all queues and pipelines have drained for that thread. Issued to the IU, and the IU performs a flush (N+1) when complete.
icbi	LSU	N/A	N/A		After the LSU generates and translates the EA, the **icbi** looks like a snooped **icbi** to the instruction fetcher.
dcbt dcbtst	LSU	N/A	N/A		
dcbz	LSU	N/A	N/A		Invalidates the L1 cache line on its way to the L2. Allocation and zero function occur at the L2 cache.
dcbst	LSU	N/A	N/A		

Table 10-1. Instruction Characteristics

Instruction	Execute			Notes	Comments
	Pipeline	Latency	Through put (ipc)		
dcbf	LSU	N/A	N/A		
slbie	LSU	N/A			Causes class-based and thread-based invalidate in both the I-ERAT and the D-ERAT
slbia	LSU	N/A			Fully invalidates the SLB, the I-ERAT and D-ERAT
tlbie	LSU	N/A			Causes index-based invalidate in both the I-ERAT and the D-ERAT Broadcast onto the EIB.
tlbiel	LSU	N/A			Causes index-based invalidate in both the I-ERAT and the D-ERAT Is not broadcast onto the EIB.
tlbsync	LSU	N/A			
slbmte	LSU	N/A		MC	
slbmfev slbmfee	LSU	N/A		MC	

Legend:
MC: Microcode
CSI: Context Synchronizing Instruction

10.2. VXU Instruction Characteristics

Table 10-2. VXU Instruction Characteristics

Instruction	Execute			Notes	Comments
	Pipeline	Latency	Through-put (ipc)		
lvebx lvehx lvewx lvlx lvlxl lvrx lvrxl lvsl lvsr lvxl	VXU load, LSU	2	1		
stvebx stvehx stvewx stvlx stvlxl stvrx stvrxl stvx stvxl	VXU load, LSU	N/A	1		
dst dstt dstst dss dssall					Not implemented (these instructions are no longer part of the architecture. They are treated as nops).
vpkpx vpkshss vpkshus vpkswss vpkuwus vpkuhum vpkuhus vpkuwum vpkuwus	permute	4	1		
vupkhpx vupkhsb vupkhsh vupklpx vupklsb vupklsh	permute	4	1		
vmrghb vmrghh vmrghw vmrglb vmrglh vmrglw	permute	4	1		
vspltb vsplth vspltw vspltisb vspltish vspltisw	permute	4	1		
vperm	permute	4	1		
vsldoi	permute	4	1		
vslo vsro	permute	4	1		
vaddubm vadduhm vadduwm vaddubs vadduhs vadduws vaddcuw vsububm vsubuhm vsubuwm vsububs vsubuhs vsubuws vsubsbs vsubshs vsubsws vsubcuw vsubsws vaddsbs vaddshs vaddsws	simple	4	1		
vavgub vavguh vavguw vagvsb vavgsh vavgsw	simple	4	1		
vand vor vxor vandc vnor	simple	4	1		
vsel	simple	4	1		
vrlb vrlh vrlw vslb vslh vslw vsl vsrb vsrh vsrw vsr vsrab vsrah vsraw	simple	4	1		
vcmpgtub vcmpgtsb vcmpgtuh vcmpgtsh vcmpgtuw vcmpgtsw vcmpgtfp	simple	4	1		
vcmpgtub. vcmpgtsb. vcmpgtuh. vcmpgtsh. vcmpgtuw. vcmpgtsw. vcmpgtfp.	simple	4	1		
vcmpequb vcmpequh vcmpequw vcmpeqfp	simple	4	1		
vcmpequb. vcmpequh. vcmpequw. vcmpeqfp.	simple	4	1		

Table 10-2. VXU Instruction Characteristics

Instruction	Execute			Notes	Comments
	Pipeline	Latency	Through-put (ipc)		
vcmpbfp vcmpgefp	simple	4	1		
vcmpbfp. vcmpgefp.	simple	4	1		
vmaxub vmaxuh vmaxuw vmaxsb vmaxsh vmaxsw vmaxfp	simple	4	1		
vminub vminuh vminuw vminsb vminsh vminsw vminfp	simple	4	1		
mtvscr	simple	N/A	N/A		Stalls at IS2 until all older instructions are complete; then, after issuing from VQ8, all younger VSU instructions stall until complete.
mfvscr	simple	N/A	N/A		Stalls at IS2 until all older instructions are complete.
vaddfp vsubfp vmaddfp vnmsubfp	float	12	1		
vrefp vrsqrtefp	estimate	14	1		
vlogefp vexptefp	float	12	1		
vrfin vrfiz vrfip vrfim vcfpsxws vcFPUxws vcuxwfp vcsxwfp	float	12	1		
vmuloub vmulouh vmulosb vmulosh vmuleub vmuleuh vmulesb vmulesh vmhaddshs vmhraddshs vmladduhm vmsumubm vmsummbm vmsumuhm vmsumuhs vmsumshm vmsumshs vsum4ubs vsum4sbs vsum4shs vsum2sws vsumsws	complex	9	1		

Legend:
MC: Microcode
CSI: Context Synchronizing Instruction

10.3. Storage Alignment Characteristics

Most storage accesses are performed without software intervention (for example, Alignment interrupt). The relative performance of storage access operations depends on their alignment. In many cases, unaligned storage accesses are handled with performance equivalent to aligned accesses. However, in some cases the hardware must break unaligned accesses into multiple internal operations. Some unaligned storage accesses cause a pipeline flush to allow a microcoded emulation of the instruction.

Table 10-3. Storage Alignment Characteristics

Operand			**Alignment**		
Type	**Size (bytes)**	**Byte Alignment**	**Within 8-B block**	**Cross 8-B boundary**	**Cross 32-B boundary**[1]
Integer load	1, 2, 4, 8	any	optimal	optimal	poor (to microcode)
Integer store	1, 2, 4, 8	any	optimal	optimal	poor (to microcode)
FP load	4, 8	not word	poor (alignment interrupt)	poor (alignment interrupt)	poor (alignment interrupt)
FP load	4, 8	word	optimal	optimal	poor (to microcode)
FP store	4, 8	not word	poor (alignment interrupt)	poor (alignment interrupt)	poor (alignment interrupt)
FP store	4, 8	word	optimal	optimal	poor (to microcode)
lmw, **stmw**	any multiple of 4B (word)	any	poor (to microcode)	poor (to microcode)	poor (to microcode)
load string word, store string word	any	any	poor (to microcode)	poor (to microcode)	poor (to microcode)
Caching-inhibited load (not from microcode)	1, 2, 4, 8, 16[3]	natural alignment	optimal	optimal	N/A
Caching-inhibited load (not from microcode)	1, 2, 4, 8	not natural alignment	poor (alignment interrupt)	poor (alignment interrupt)	poor (alignment interrupt)
Caching-inhibited store (not from microcode)	1, 2, 4, 8, 16[3]	natural alignment	optimal	optimal	N/A
Caching-inhibited store (not from microcode)	1, 2, 4, 8	not natural alignment	poor (alignment interrupt)	poor (alignment interrupt)	poor (alignment interrupt)
Caching-inhibited load or store from microcode	1, 2, 4, 8, 16	any	poor (alignment interrupt)	poor (alignment interrupt)	poor (alignment interrupt)
VXU load (except **lvlx[l]**, **lvrx[l]**)	16	any	optimal	optimal	N/A[3]
VXU store (except **stvlx[l]**, **stvrx[l]**)	16	any	optimal	optimal	N/A[3]
VXU load/store left (**lvlx[l]**, **stvlx[l]**)	any	any	optimal	optimal	N/A[4]
VXU load/store right (**lvrx[l]**, **stvrx[l]**)	any	not QW aligned	optimal	optimal	N/A[4]
VXU load/store right (**lvrx[l]**, **stvrx[l]**)	any	QW aligned[2]	optimal	optimal	N/A[4]

1. Ops crossing 4-K, 64-K, or Segment boundaries behave the same as crossing a 32-byte boundary (or an 8-byte boundary in DABR or TDABR mode).
2. For Load Vector Right and Store Vector Right that are quadword aligned, no attempt is made to access storage. On a load the data returned is 0, on a store the op becomes a no-op.
3. Regular VXU ops are assumed to be aligned to the 16-byte boundary.
4. Due to the way these ops are defined, they do not cross a 32-byte boundary (or an 8-byte boundary in DABR or TDABR mode) (data is accessed up to a 16-byte boundary).

The PPU initiates an Alignment interrupt in the following instances:

- The operand of a floating-point load or store is not aligned, or crosses a virtual page boundary.
- The operand of **lmw**, **stmw**, **lwarx**, **ldarx**, **stwcx**., or **stdcx**. is not aligned.
- The instruction is **lmw**, **stmw**, **lswi**, **lswx**, **stswi**, or **stswx**, and the operand is in storage that is caching-inhibited.
- The operand of a load or store is not aligned and is in storage that is caching-inhibited.
- The operand of **lwarx**, **ldarx**, **stwcx**., or **stdcx**. is in storage that is caching-inhibited.
- The instruction is **lha**, **lhau**, **lhaux**, **lhax**, **lwa**, **lwaux**, or **lwax**, and the operand is in storage that is caching-inhibited.
- The instruction is **dcbz** and the operand is in storage that is caching-inhibited or the L1 Dcache is disabled via HID4[en_dcway] = '0000'.
- The instruction is **lfd**, **lfdx**, **lfdu**, **lfdux**, **stfd**, **stfdx**, **stfdu**, or **stfdux**, and the operand is in storage that is caching-inhibited, and the operand is word-aligned, but is not double-word aligned.

Programming Note:

Any load or store instruction that does not cause an alignment interrupt, but crosses a virtual page boundary is flushed and sent to microcode engine. This lowers the performance of such instructions.
However, misalignments that are handled by microcode can be forced to cause an Alignment interrupt by setting HID4[force_ai] to '1'.

10.4. Microcoded Instructions

Instructions coming from the IBufs are decoded in the ID1 stages of the pipeline. If there are no stalls, then most instructions flow through the pipeline uninterrupted. Some instructions which are either too complex or require too many system resources to implement (such as load string instructions) are microcoded. This means that they are split into several simpler instructions (microwords).

A minimum of 11 cycles is required before the first instruction is received from the microcode ROM, so microcoded instructions should be avoided if possible. Most microcoded instructions are decoded into two or three simple PowerPC instructions, and can be avoided in most cases. The microcoded instructions are typically decomposed into integer and load/store operations with a dependency between them. Although most microcoded PowerPC instructions are decoded into only a few simple instructions, it is important to keep in mind that there are typically dependencies between the internal operations of the microcode, which generate stalls at the issue stage. Replacing the microcoded instructions with PowerPC instructions not only avoids stalling for the ROM, but gives more latitude in scheduling instructions to avoid stalls, as well as potentially improving multithreaded performance.

Some microcoded instructions are more complex than just a few PowerPC instructions, and some instructions are only microcoded in certain conditions, so replacement may not be possible. Instructions that are always decoded into microcode are referred to as unconditionally microcoded. Instructions that are microcoded only under specific conditions are referred to as conditionally microcoded. The only instructions that are conditionally microcoded are unaligned loads and stores that would not be microcoded if they were aligned.

Programming Note:

A microcoded instruction is a single instruction and must execute atomically. Therefore, an asynchronous interrupt cannot be taken while a microcoded instruction is executing. This is another reason why microcoded instruction should be avoided.

10.4.1. Unconditionally Microcoded instructions

Instructions which are difficult to implement in hardware or are infrequently executed may be split into microcode instructions. Instructions which are always microcoded can be summarized by the following list:

- Shifts and rotates which read the shift amount from a register instead of from an immediate field
- Load/store algebraic
- Load/store strings and multiples
- Several CR recording instructions (“Rc” = ‘1’)
- Other instructions

10.4.2. Conditionally Microcoded Instructions

The L1 DCache is physically implemented with 32-byte sectors, and a conditionally microcoded Load or Store instruction which attempts to perform a L1 DCache access that crosses a 32-byte boundary must be split into several instructions. When one of these misaligned load or stores first attempts to access the L1 DCache, the misalignment is detected and the pipeline is flushed when the instruction reaches the EX7 stage. The flushed load or store is then refetched, and converted to microcode at the decode stage, and is split into the appropriate loads or stores, as well as any instructions needed to merge the values together into a single register.

Doubleword integer loads which cross a 32-byte alignment boundary are first attempted as two word-sized loads or stores. If these still cross the 32-byte boundary, they are flushed and attempted again at byte granularity. The word and halfword integer loads behaves similarly.

Doubleword floating-point loads and stores which are aligned to a word boundary, but not a doubleword boundary, are handled in microcode. If these loads or stores are not word aligned, or if they cross a virtual page boundary, a PowerPC Alignment interrupt is taken.

All of the Conditionally Microcoded instructions are Loads and Stores.

Programming Note:

Integer loads and stores which are misaligned but do not cross a 32-byte boundary are not converted into microcode and have the same performance characteristics as aligned loads and stores.

10.5. Pipeline Hazards and Performance Considerations

10.5.1. Instruction Issue

The processor supports dual-issue of instructions at both pipeline stages at which instructions are issued, IS2 and VQ8 (see *Section 6.2 PPU Pipeline* on page 24). There are several restrictions that limit which instructions can be issued together.

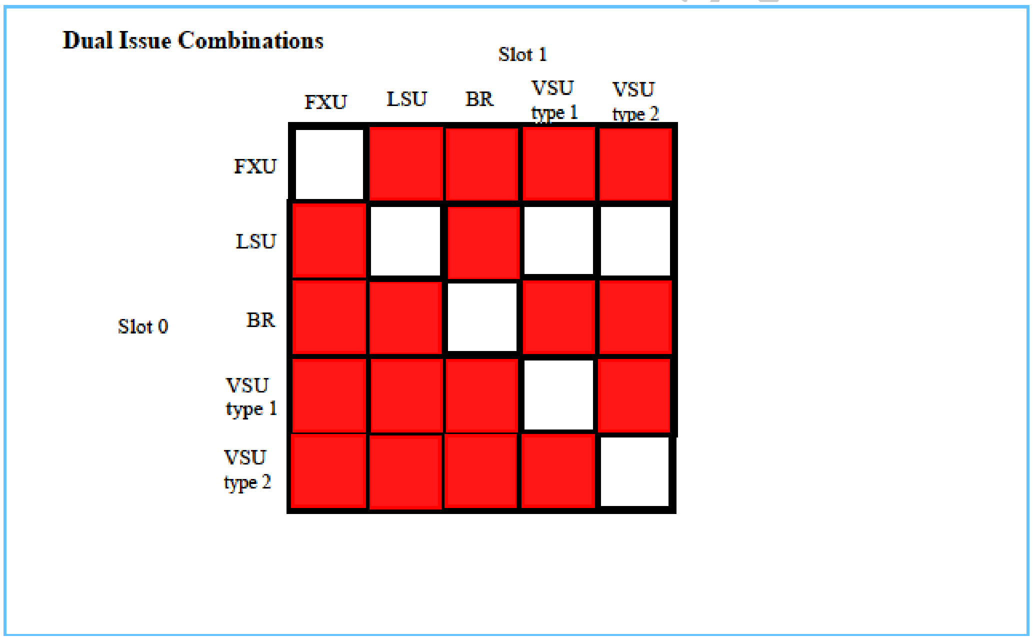
Figure 10-1 demonstrates valid issue combinations. The instruction in slot 0 (the older instruction) is shown on the left vertical axis and the instruction in slot 1 (the younger instruction) is shown on the top horizontal axis. The red squares demonstrate valid dual-issue combinations, and the white squares indicate illegal combinations.

The VXU and FPU instructions are categorized into two groups. The first group, type 1, includes VXU Simple, VXU Complex, VXU Float, and FPU Arithmetic instructions. The second group, type 2, includes VXU Load, VXU Store, VXU Permute, FPU Load, and FPU Store instructions.

Also, some instructions are classified in more than one group. For example, all fixed-point store and load-with-update instructions require both the FXU and the LSU in order to execute. In this case, both rows or columns for FXU and LSU must be checked to see if an issue conflict will occur. The classification of each PowerPC Architecture instruction can be found in *Table 10-1* on page 88 and *Table 10-2* on page 94 (in the column labeled "Pipeline").

Note: All stall conditions prevent dual issue. See *Section 10.5.2 IS2 Stall Conditions* on page 100 and *Section 10.5.3 VQ8 Stall Conditions* on page 101 for comprehensive lists of stall conditions.

Figure 10-1. Dual Issue Combinations



Slot 0 \ Slot 1	FXU	LSU	BR	VSU type 1	VSU type 2
FXU	⬜	🟥	🟥	🟥	🟥
LSU	🟥	⬜	🟥	⬜	⬜
BR	🟥	🟥	⬜	🟥	🟥
VSU type 1	🟥	🟥	🟥	⬜	🟥
VSU type 2	🟥	🟥	🟥	🟥	⬜

Only one branch instruction, one instruction that updates the CR, or one instruction that accesses the CR can be issued per cycle. Some fixed-point and floating-point operations update the CR (Rc = '1'). The store conditional operations also update the CR. Therefore, the following combinations cannot be dual issued:

- Fixed-point (Rc = '1') and branch
- Fixed-point (Rc = '1') and VSU type 1 (Rc = '1')
- VSU type 1 (Rc = '1') and fixed-point (Rc = '1')
- VSU type 1 (Rc = '1') and store conditional

- VSU type 1 (Rc = ‘1’) and branch
- Store conditional and branch
- Store conditional and VSU type 1 (Rc = ‘1’)

There are several specific cases that prevent the dual issue of instructions at the IS2 stage:

- A nonpipelined instruction is encountered in slot 0 (the older instruction in a dispatch pair)
- A context-synchronizing instruction is encountered in either slot
- A microcoded instruction is encountered
- The processor is single-stepping due to MSR[FE0] or MSR[FE1] being set to '1'
- The VSU Issue Queue (VIQ) is single-stepping due to an internal flush (as the result of a denormalized operand, for example)

There are several specific cases that prevent the dual issue of instructions at the VQ8 stage:

- A VXU float instruction (except for a reciprocal-estimate operation) cannot be issued in the two cycles following a **vrefp** and **vrsqrtefp**.
- No FPU instruction (including loads and stores) can be issued after a FPU **fdiv** or **fsqrt** instruction until the **fdiv** or **fsqrt** finishes. Vector/SIMD Multimedia Extension instructions are also stalled.
- A younger VSU load or store can be issued in the same cycle as any other VSU type 1 instruction, but an older VSU load or store cannot.

10.5.2. IS2 Stall Conditions

The following conditions can cause a stall at the IS2 stall point:

- Stall dispatch slot 1 if slot 0 is stalled
- Read-After-Write (RAW) dependency
  - 1-cycle bubble for an FXU operation dependent on an FXU or LSU operation
  - 4-cycle bubble for a load operation dependent on an FXU or LSU operation
  - 4-cycle bubble for when the address of a store operation is dependent on an FXU or LSU operation
  - When the data of a store operation is dependent on an FXU or LSU operation, the data must be dispatched at least two cycles after sourcing from the FXU or LSU operation. If dispatched less than two cycles after, then a 4-cycle bubble occurs.
  - 1-cycle bubble between two instructions that access the Fixed-Point Exception Register (XER) (Two exceptions to this are an instruction that sets the XER followed by an mfxer or followed by a store conditional instruction)
  - CTR decrement instruction followed by a CTR-using instruction (1 bubble between groups)
  - CR fields. A dirty-bit scheme is used for long-latency instructions that update the CR, to flush CR-dependent operations until the CR producer is completed.
- Write-After-Write (WAW) dependency can only occur for instructions dependent on a load that misses the L1 DCache. When this occurs, the dependent instruction is flushed rather than stalled.
  When a CR WAW dependency occurs, the dependent instruction is also flushed rather than stalled when the producer takes longer than normal to update the CR. For some CR WAW cases, the flush is masked to improve performance.
- Write-After-Read (WAR) dependencies can only occur for store conditional instructions followed by a write to the XER. If an operation writes the XER while the store conditional instruction is pending, then that operation is flushed.
- Route collision within a dispatch group (as described in *Section 10.5.1 Instruction Issue*)
  - If both instructions are routed to the FXU, then the second instruction will stall for 1 cycle.
  - If both instructions are routed to the BRU, then the second instruction will stall for 1 cycle.
  - If both instructions are routed to the LSU, then the second instruction will stall for 1 cycle.

  - If both instructions access the Condition Register (CR) (read or write), then the second instruction will stall for 1 cycle.
  - If both instructions access the LR (read or write), then the second instruction will stall for 1 cycle.
  - If any LSU operation is in slot 0, any VXU or FPU operation in slot 1 will stall for one cycle (regardless of whether it is dependent)

  Programming Note: The compiler should reverse the order of any VXU or FPU load followed immediately by a nondependent VXU or FPU instruction to avoid this stall condition.
- If a VQ8 stall occurs, IS2 will stall two cycles later if and only if there is a VXU or FPU instruction in IS1 or IS2.
- Execution Unit (XU) Nonpipelined instructions
- Context-synchronizing instruction (CSI) stall until all previous instructions have completed before issuing.
- LSU (external) stall requests can arise when any of these are required:
  - L1 DCache directory or array write
  - ERAT write
  - CSI or nonpipelined operations
  - Recycling a load miss
- Store pacing for microcoded stores

10.5.3. VQ8 Stall Conditions

The following conditions can cause a stall in the VIQ at cycle VQ8. The number of cycles a stall must be raised for RAW, WAW, WPC, and RFC conditions can be inferred from *Figure 6-9. VSU Dataflow Diagram* on page 42. The VXU and FPU instructions are categorized into two groups. The first group, type 1, includes VXU Simple, VXU Complex, VXU Float, and FPU Arithmetic instructions. The second group, type 2, includes VXU Load, VXU Store, VXU Permute, FPU Load, and FPU Store instructions.

- Stall dispatch slot 1 if slot 0 stalled
- RAW or WAW dependency
- Resource conflict. Two operations want to issue to the same group (as defined above). The second instruction will stall for one cycle.
- Write-port conflict (WPC). If two instructions would collide when writing results if issued together, the second instruction will stall until no such conflict exists. The speculative nature of WPC stalling may lead to unnecessary stalling.
- Register file conflict (RFC). Two instructions cannot write the same address in the GPR Register File at the same time. The second instruction will stall until no such conflict exists.
- VSU load recycles. Stall to insert back into pipeline. Load recycles are speculative, and always take one cycle longer than necessary.
- Table operations (**vrefp**, **vrsqrtefp**). These operations always incur a 1-cycle bubble after issuing; they still allow a younger slot-1 operation to dual issue.
- In FPU-enabled invalid operation exception or a zero divide exception mode (set in the FPSCR), floating-point stores are stalled at VQ8 until the VSU pipeline is drained, unless the store depends on a load in the VMQ.
- Nonpipelined operations (**fdiv**, **fsqrt**). Stall until the operation is finished.
- Denorm mode. Each instruction in the VSU Denorm Queue is single-stepped to completion.
- Single-step mode (for hardware-debug)
- Single-issue mode stalls dispatch slot 1 (for hardware-debug)
- **mffs** — Stall until all issued instructions have completed.
- **mtvscr** — Stall all subsequent instructions at VQ8 until **mtvscr** has completed.

10.5.4. Instruction Behavior

10.5.4.1. Simple Integer Instructions

Simple integer instructions are fully pipelined and execute in two cycles (there is a pipeline bubble of one cycle when a direct dependency exists).

10.5.4.2. Complex Integer (Nonpipelined) Instructions

The Complex XU pipeline is used to execute integer multiplies and divides. Because these instructions are not fully pipelined, they are implemented as double-issue instructions, as described in *Section 10.5.8 Double-Issue and Context Synchronizing Instructions* on page 103. The latencies and throughputs of the different forms of integer multiplies and divides are listed in *Table 10-1. Instruction Characteristics* on page 88.

10.5.5. CR Dependencies

A CR-dependent instruction does not issue in the same cycle as a branch (irrespective of any dependency) to avoid a resource collision. A CR-dependent instruction can issue in back-to-back cycles with a dependent branch. Most CR dependencies do not generate a stall. For example a **cmp**, **cmpi**, **cmpl**, or **cmpli** followed in the next cycle by a branch that has a CR dependency does not generate a stall. The result of the compare can be forwarded to the dependant branch without generating a stall. However, a few cases can cause the pipeline to stall or flush on a CR dependency:

- When a **stdcx.** or **stwcx.** instruction is executed, the next instruction to read CR field 0 is flushed and held at dispatch until the **stdcx.** or **stwcx.** instruction is complete. This also applies to the next instruction to write CR field 0 if it is a “long latency” or “logical” type of instruction.
- All VXU and FPU instructions that set a CR field also set an internal “dirty” bit while executing, which indicates that they intend to write a specific CR (one dirty bit exists for each CR field). When the instruction finishes executing and the result is written back, the dirty flag is cleared.
  Any subsequent instruction that attempts to read a CR field with its dirty bit set is flushed at EX7 and refetched. If the dirty bit is set and any VXU, FPU, or **stwcx.** instruction causes a write-after-write condition, it is flushed. (The only WAW cases that do not cause a flush are the “fast CR” types.) Otherwise, the CR is updated correctly by the hardware without a flush.
- The **mfcr** instruction is treated as a double-issue instruction (see *Section 10.5.8 Double-Issue and Context Synchronizing Instructions* on page 103).

10.5.6. Special Purpose Register Hazards

The most common **mtspr** and **mfspr** instructions read data from the Link (LR) and Count (CTR) registers. Because of the frequency of execution, these Special Purpose Register (SPR) instructions are optimized so that an instruction that sets the LR can be immediately followed by an **mfspr** LR with no stall penalty. Similarly, an **mtspr** LR does not generate a stall if the following instruction reads the LR. The behavior for the CTR register is similar. The **mfspr** LR and CTR instructions, themselves, require multiple cycles to execute since they are double-issue instructions (see *Section 10.5.8 Double-Issue and Context Synchronizing Instructions* on page 103).

The **mtspr** instruction is fully pipelined and does not cause stalls or flushes. The **mtmsr**(**d**) L = 0 instruction (or **mtspr** MSR instruction) requires that all currently executing instructions complete before the instruction can be executed. Because of this, **mtmsrd** is microcoded as: sync L = 0; **mtmsr**(**d**) L = 0; sync L = 0; **isync**. The **mtmsr**(**d**) L = 1 instruction is not microcoded, but is execution synchronizing (see the Move to Machine State Register instructions in *Book III: PowerPC Operating Environment Architecture*). Hence, it executes as a normal pipelined instruction unless an External or Decrementer exception is pending with MSR[EE] = '0'. In this case, the interrupt is taken and all younger instructions are flushed.

Because most SPRs are located in physically distant places on the chip, the time to access them varies from SPR to SPR. To simplify timing, the **mfspr** instructions are implemented as double-issue instructions (with the exception of LR and CTR). See *Section 10.5.8 Double-Issue and Context Synchronizing Instructions* on page 103 for details about double-issue instructions. Following the **mfspr**, instruction issue is stalled (for both threads) until the SPR value is finished being read.

10.5.7. XER Dependencies

The XER is treated as a single contention point for purposes of stalling. Hence, instructions that are dependent on the XER may be stalled by an XER setting instruction, even if they use different fields within the XER.

10.5.8. Double-Issue and Context Synchronizing Instructions

Some instructions, such as multiply, divide, and slow **mfspr** instructions, need to make sure that all younger instructions stall until they have completed executing. These kinds of instructions are implemented as double-issue instructions, which means that they are issued twice to the execution units. The first time the instruction issues, it issues as a regular instruction, except that it also remains in the IS2 stage, blocking issue until it completes. The stalled instruction can leave the issue stage when its dependency is satisfied on the double-issue instruction, so the instruction issues again and execution proceeds normally.

All **mfspr** instructions are double-issued. The second issue of the CTR and LR **mfspr** instructions occurs two cycles after the first issue. Other SPR reads are variable and typically require between ten and thirty cycles before the second issue of the instruction.

Because double-issue instruction must be executed atomically, asynchronous interrupts cannot be taken between the first and second issues of the instruction. In some extreme cases, the worst-case interrupt latency can become very large.

Instructions that are defined as Context Synchronizing, such as **attn**, **sc**, **mtctrl**, **mtvscr**, **mfvscr**, **sync**, **isync**, **hrfid**, **tlbie**(**l**), and **rfid** (described in the Context Synchronization section of *Book III: PowerPC Operating Environment Architecture*) are pipeline-draining instructions. Before issuing, these instructions block issue until all of the instructions in the processor have completed (for both threads), including instructions in the load miss queue (LMQ) and the store queue (STQ), and any outstanding MMU operations.

In the case of **isync**, **attn**, **rfid**, **hrfid**, **sc**, and **mtctrl**, when the instruction reaches EX7, the pipeline is flushed in order to clear out any stale instructions that may have been fetched from the L1 ICache.

In the case of **sync** and **tlbie**(**l**), after the instruction is issued, all subsequent instruction issue stalls until the **sync** and **tlbie**(**l**) are issued from the store queue to the memory subsystem.

10.5.9. Store Instruction Handling and the Store Queue

Stores that are issued to the LSU make two passes through the LSU pipeline.
When the store is issued, the D-ERAT and cache tags are first checked to determine if the store was a hit to the cache, and its location in the cache (since the L1 DCache is set associative). However, the store does not write its data into the L1 DCache, because it still considered a speculative instruction, and it is not known until the EX7 stage (the flush point) whether or not the store may be flushed. The store is placed into the STQ.

When the store has become the oldest instruction in the machine, it is marked as being able to complete (and is moved into a single entry buffer after the STQ). From this point, the store logic looks for an unused cycle to write the L1 Dcache. An unused cycle occurs either when a store instruction is being issued or when there is no operation being issued. (The store write must wait if a load is being issued). Once this slot is found, then if the L1 Dcache needs updating, the L1 Dcache is written, and the store is sent to the L2.
This second pass through the pipeline causes the L1 DCache data array to become busy, but not the tag array. Therefore, the data can be written in parallel while another store is being issued and accessing the L1 DCache tags.

The L1 DCache is a write-through, non write-allocate cache, so stores are always sent to the L2, but are only written to the L1 when the appropriate line is in the L1 DCache.

Besides store instructions, the STQ is also responsible for holding several other instructions that relate to memory access, such as **sync**, **eieio**, **stwcx.** The behavior of these instructions is discussed in their own sections.

10.5.10. Load Miss Handling

Loads that miss the L1 DCache enter the Load Miss Queue (LMQ) for processing in the L2 and memory system. Data is returned from the L2 in 32-byte beats on four consecutive cycles. The first cycle contains the critical section of data, which is formatted and sent to the register file. The L1 DCache is occupied for two consecutive cycles while the reload is written back to the cache, half a line at a time. The tag array is then updated on the next cycle. All instruction issue is stalled during the two cycles while the reload is written back to the DCache and also during the data forwarding cycle. In addition, no instructions can be recycled during this time. The LMQ entry can then be used seven cycles after the last beat of data returns from the L2 to handle another request.

Instructions that are dependent on a load are issued speculatively assuming a load hit. If it is later determined that the load missed the L1 DCache, younger instructions and any instructions that are dependent on the load are flushed, refetched, and held at dispatch until the load data has been returned. This behavior allows the PPE to send multiple overlapping loads out to the L2 without stalling if they are dependent. In multithreading mode, this behavior allows load misses from one thread to occur without stalling the other thread. Reloads returning from the L2 have the highest priority for accessing the L1 DCache, followed by recycled loads and then load or store instructions from the issue stage.

Programming Note:

In general, write-after-write (WAW) hazards do not cause penalties in the PPE. However, if the target register for an instruction is the same as the target register for an outstanding load miss, the new instruction is flushed, refetched, and held at dispatch until the older load writes its result to the register file, in order to avoid having stale values written back to the register file.

Programming Note:

Some old software performed prefetching by executing load instructions to bring cache lines into the L1 DCache, and the result of the loads were discarded into a ‘garbage’ GPR. This would not be effective on the PPE because this WAW condition would effectively serialize the prefetch. This is not a problem if DCBT instructions are used instead.

10.5.11. Load-Hit-Reload and Load-Hit-Store

The load miss queue (LMQ) is used to hold loads that have missed the L1 cache and have a request pending in the L2. It is also used to hold loads when a load-hit-reload (LHR) or a load-hit-store (LHS) condition occurs.

Load-Hit-Reload

When a reload for a load miss is pending and a younger load attempts to read the cache line at a similar address as the pending reload, an LHR condition occurs and the new load is allocated an LMQ entry and marked as an LHR. No request is made to the L2, because the cache line has already been requested. The LHR simply waits for the original request to return. When the line is returned, the new load is recycled through the pipeline and is given the opportunity to read the data from the L1 DCache (as a cache hit).

Programming Note:

If a load miss is followed by seven loads to the same cache line, these seven loads will all be marked as LHR and will occupy all LMQ entries, preventing any independent load misses from being sent to the L2.

Load-Hit-Store

The load-hit store (LHS) condition occurs when a load request is made to a store to a similar address in the processor’s STQ, regardless of the L1 DCache hit/miss of the load request. Because the STQ in the processor holds stores that have not yet written into the cache, the load must wait in an LMQ until the store has written its data into the cache. Loads that are marked as LHS can be recycled after all older stores have completed. No forwarding of data between the STQ and loads is performed (bypassing, or store forwarding mechanism).

During recycling, instruction issue is stalled while the recycled load reenters the LSU pipeline. The LMQ entry for the recycled load is not deallocated until the load is determined to be a hit in the L1 DCache.
A recycled load-hit-store (LHS) load may hit or may miss the DCache, because of the store operation under the write-through, non write-allocate cache policy.
A hit is not guaranteed for the recycled load-hit-reload (LHR) load, because of the alias cases that can exist (see *Section 10.5.11.1 Load-Hit-Reload and Load-Hit-Store Aliasing* for more detail).
The alias cases may also occur for the LHS. In an alias case, a load may be falsely marked as an LHR or LHS.

A miss case of a recycled load means that when the load is recycled and checks the L1 DCache, it will not find the data, and will need to make a request to the L2 when the load reenters the LMQ in EX6.
Recycled loads can never be marked as LHR or LHS again, since by the nature of the mechanism, the load must be older than the other entries in the LMQ and STQ.

Because of the weakly consistent storage access ordering of the PowerPC Architecture (see *Book II: PowerPC Virtual Environment Architecture*), loads in the LMQ or stores in the STQ from another thread are not required to be compared.
Entries in the LMQ and STQ that are from the other thread are not required to cause an LHR or LHS.
However, for performance reason to avoid duplicated reload, a missed load is compared to both threads.

10.5.11.1. Load-Hit-Reload and Load-Hit-Store Aliasing

Because of the high-frequency design, a full 42-bit real address comparison is not performed when checking for LHR and LHS conditions. In particular, bits EA[0:51] (RA[22:51]) are not compared. Because of this, some alias cases exist that can cause a load to be marked as an LHR or LHS. The false aliasing does not affect the functional correctness of execution, but can cause a load to be recycled when it is not really needed.

If the load misses the L1 DCache, then a comparison is performed between RA[52:56] of the load miss and the entries of the LMQ (for LHR) or the STQ (for LHS).
For a load that hits L1 DCache, an LHS comparison is performed between RA[52:61] (for DD3) or RA[52:59] (for DD2) of the load and the entries of the STQ that are from the same thread.

Cache operations (DCBT, DCBZ, and so forth) are always compared with RA[52:56].

10.5.11.2. Performance Considerations on Load-Hit-Store Hazards

Since LHS conditions incur considerably large penalties as pipeline hazards, programmers are advised to avoid the occurrence of LHS conditions as much as possible.

True LHS Hazards

Basically, a load instruction should be far enough from the dependent store instruction in order to avoid a true LHS hazard, but it may not be generally possible.
Here are some typical examples which may incur true LHS hazards.

- Stack operations

  In general, stack operations may involve a store and a load to the same address in a short period. Here are some tips to minimize such situations.

  - Avoid small functions whose execution time is short with stack operations at its entry and exit.
  - Pass arguments in the registers as much as possible to avoid arguments-passing via the stack.
- Volatile/global variables

  Accesses to volatile or global variables usually require memory accesses which tend to incur true LHS hazards. Avoid excessive use of volatile or global variables, if possible.
- Data transfer between different types of register files

Data transfers between different types of register files (GPR, FPR and VPR) must be performed via memory. Such transfers should be avoided if possible because they will cause true LHS hazards. Typical examples:

- Type cast from `int` to `float` and vice versa
- Mixed operation of vectors and scalars

False LHS Hazards

As described in *Section 10.5.11.1 Load-Hit-Reload and Load-Hit-Store Aliasing*, false detections of LHS may occur due to aliases. This results in "false" LHS hazards. Programmers should avoid the cause of LHS aliases as shown in the following examples.

- Upper EA alias example:
  store to 0x1000
  load from 0x2000
- Lower EA alias example:
  store byte to 0x4
  load byte from 0x7

Programming Note:

The performance monitor facility may be useful to analyze the degree and frequency of the LHS stalls in a program.

10.5.12. Data Address Translation

The data-side ERAT (D-ERAT) is a 64-entry, 2-way set-associative effective-to-real-address translation cache. Each D-ERAT entry holds the effective-to-real address translation for an aligned 4-KB area of memory. When using 4-KB pages, each D-ERAT entry holds the information for a single page, and when using large pages, each ERAT entry holds a 4-KB section of the page, meaning that a large page can occupy several entries in the D-ERAT.

The D-ERAT is accessed in parallel with the L1 DCache tags, and a comparison is performed at the end to determine if the access was a hit or not. The operation of the D-ERAT is very similar to the I-ERAT.

When an instruction misses the D-ERAT, it is held in an ERAT miss queue while a request is sent to the MMU for translation. This invokes an 11-cycle penalty in order to translate the address in the MMU. The instruction following the ERAT miss is flushed, refetched, and held at dispatch (while the other thread is given all of the dispatch slots, similar to a L1 DCache miss). If this translation hits in the TLB, the ERAT entry is reloaded and the fetch is attempted again. This instruction will see approximately a 27-cycle penalty when an ERAT miss occurs in order to translate the address in the MMU, reload the ERAT, and recycle the instruction.

If the request misses the TLB, then a hardware tablewalk is initiated if LPCR[TL] = ‘0’. The tablewalk has a variable penalty depending on whether the page table entry groups (PTEGs) are cached in the L2. If LPCR[TL] = ‘1’, then a page fault is generated so that software can reload the TLB entry itself. In general, a hardware tablewalk can require several hundred cycles or more. In the best case, where the PTE entry is found in the cache and is the first entry of the PTEG, the latency is approximately 64 cycles (including the MMU overhead). If the entry is found in an even primary PTEG entry (0, 2, 4, or 6), then the latency is less than if it is found in an odd PTEG entry (1, 3, 5, 7). This is because a second L2 access is required to search for odd PTEG entries. Software can organize the PTEG entries with this knowledge to reduce TLB miss penalties. The secondary PTEG search works similarly. The order of searching is as follows (each line represents an L2 request, which represents approximately an additional 20 cycles of latency):

- Primary PTEG 0, 2, 4, 6
- Primary PTEG 1, 3, 5, 7
- Secondary PTEG 0, 2, 4, 6

- Secondary PTEG 1, 3, 5, 7

Only one outstanding ERAT miss request to the MMU is allowed at any given time. In multithreading mode, if a thread misses the D-ERAT while another thread has a pending D-ERAT miss, the new instruction is flushed, refetched, and held at dispatch until the ERAT entry from the first miss is reloaded into the ERAT.

The MMU services one translation request at a time. To prevent unnecessary delays, the MMU does not speculatively perform tablewalks, since this can potentially starve a valid request from the other thread and pollute the TLB (and the L2). The MMU services a speculative request if it hits the TLB, but tablewalks are only performed for demand requests, as they are nonspeculative.

10.5.12.1. Performance Considerations on D-ERAT

As described above, the D-ERAT is a 64-entry, 2-way set-associative cache, and each of its entries corresponds to an aligned 4-KB area of memory even when using large pages.

Programmers should be aware that significant performance degradation due to the "D-ERAT thrashing" may occur in the following conditions.

- Three or more separate areas are accessed by rotation.
- Locations of the areas accessed concurrently have the same offset (in 4-KB granule) with respect to 128-KB boundaries.

Example:

```
# define BUFF_SIZE 32*1024

int InBuf0[BUFF_SIZE];  // 128-KB buffers
int InBuf1[BUFF_SIZE];
int OutBuf[BUFF_SIZE];

for (int i=0; i<BUFF_SIZE; i++) {
   OutBuf[i] = InBuf0[i] + InBuf1[i];  // D-ERAT thrashing will occur
}
```

Programming Note:

The performance monitor facility may be useful to analyze the degree and frequency of the D-ERAT miss stalls in a program.

11. Architecture Compliance

This section describes how the PPE complies with and deviates from the PowerPC Architecture. The PPE is fully compliant with Version 2.02 of the PowerPC with only the few minor exceptions documented in this section.

11.1. Book I Defined Instructions

This section describes which *Book I: PowerPC User Instruction Set Architecture* instructions are implemented.

The following optional user-mode instructions are implemented:

- **fsqrt(.)** Floating Square Root A-form
- **fsqrts(.)** Floating Square Root Single A-form
- **fres(.)** Floating Reciprocal Estimate Single A-form
- **frsqrte(.)** Floating Reciprocal Square Root Estimate A-form
- **fsel(.)** Floating Select A-form
- **mtocrf** Move To One Condition Register Field XFX-form
- **mfocrf** Move From One Condition Register Field XFX-form

The following user-mode instruction newly defined in the version 2.02 of *Book I: PowerPC User Instruction Set Architecture* is not implemented. Use of this instruction causes an Illegal Instruction type of Program Interrupt. This is a deviation from the version 2.02 of the PowerPC Architecture for this implementation.

- **popcntb** Population Count Bytes X-form

The following optional user-mode instructions newly defined in the version 2.02 of *Book I: PowerPC User Instruction Set Architecture* are not implemented. Use of these instructions causes an Illegal Instruction type of Program Interrupt:

- **fre(.)** Floating Reciprocal Estimate A-form
- **frsqrtes(.)** Floating-Point Reciprocal Square Root Estimate Single A-form

The following optional facilities defined in *Book I: PowerPC User Instruction Set Architecture* are not implemented.

- Little endian mode

The following obsolete instructions (or instruction forms) are not implemented and results in an Illegal Instruction type of Program interrupt:

- **mcrxr** Move from Condition Register to XER Register
- **bccbr** Branch Conditional to CBR

11.2. PPE-Specific Instructions

The following instructions are implemented that are not defined in the PowerPC Architecture.

- **ldbrx** Load Doubleword Byte Reverse Indexed X-form
- **sdbrx** Store Doubleword Byte Reverse Indexed X-form

Programming Note:

Because these instructions are not part of the architecture, they should be considered highly implementation specific. Any code that uses these instructions should keep this in mind when portability considerations are a concern.

Load Doubleword Byte-Reverse Indexed X-form

ldbrx RT, RA, RB

31	RT	RA	RB	532	/
0	6	11	16	21	31

```
if RA = 0 then b ← 0
else           b ← (RA)
EA ← b + (RB)
RT ← MEM(EA+7, 1) || MEM(EA+6, 1) || MEM(EA+5, 1) || MEM(EA+4, 1) ||
     MEM(EA+3, 1) || MEM(EA+2, 1) || MEM(EA+1, 1) || MEM(EA, 1)
```

Let the effective address (EA) be the sum (RA|0)+(RB).
Bits 0:7 of the doubleword in storage addressed by EA are loaded into RT$_{56:63}$.
Bits 8:15 of the doubleword in storage addressed by EA are loaded into RT$_{48:55}$.
Bits 16:23 of the doubleword in storage addressed by EA are loaded into RT$_{40:47}$.
Bits 24:31 of the doubleword in storage addressed by EA are loaded into RT$_{32:39}$.
Bits 32:39 of the doubleword in storage addressed by EA are loaded into RT$_{24:31}$.
Bits 40:47 of the doubleword in storage addressed by EA are loaded into RT$_{16:23}$.
Bits 48:55 of the doubleword in storage addressed by EA are loaded into RT$_{8:15}$.
Bits 56:63 of the doubleword in storage addressed by EA are loaded into RT$_{0:7}$.

Special Registers Altered:

None.

Store Doubleword Byte-Reverse Indexed X-form

stdbrx RS, RA, RB

31	RS	RA	RB	660	/
0	6	11	16	21	31

```
if RA = 0 then b ← 0
else           b ← (RA)
EA ← b + (RB)
MEM(EA, 8) ← $(RS)_{56:63}$ || $(RS)_{48:55}$ || $(RS)_{40:47}$ || $(RS)_{32:39}$ || $(RS)_{24:31}$ || $(RS)_{16:23}$ || $(RS)_{8:15}$ || $(RS)_{0:7}$
```

Let the effective address (EA) be the sum (RA|0)+(RB).
(RS)$_{56:63}$ are stored into bits 0:7 of the doubleword in storage addressed by EA.
(RS)$_{48:55}$ are stored into bits 8:15 of the doubleword in storage addressed by EA.
(RS)$_{40:47}$ are stored into bits 16:23 of the doubleword in storage addressed by EA.
(RS)$_{32:39}$ are stored into bits 24:31 of the doubleword in storage addressed by EA.
(RS)$_{24:31}$ are stored into bits 32:39 of the doubleword in storage addressed by EA.
(RS)$_{16:23}$ are stored into bits 40:47 of the doubleword in storage addressed by EA.
(RS)$_{8:15}$ are stored into bits 48:55 of the doubleword in storage addressed by EA.
(RS)$_{0:7}$ are stored into bits 56:63 of the doubleword in storage addressed by EA.

Special Registers Altered:

None.

11.3. Floating-Point Exception Mode

There are four floating-point exception modes defined by the PowerPC Architecture, that are specified by MSR[52] and MSR[55]:

1. Ignore Exceptions Mode
2. Imprecise Nonrecoverable Mode
3. Imprecise Recoverable Mode
4. Precise Mode

The processor supports two of these modes:

- Ignore Exceptions Mode
- Precise Mode

If either Recoverable or Nonrecoverable Imprecise exception mode is enabled, the PPE operates as if in Precise mode. See *Section 5.2.2.1 Machine State Register (MSR)* on page 20 for more information.

Caution: There is a significant performance degradation in Precise Mode because the processor “single-steps” all instructions when set in this mode.

11.4. FPSCR[NI] and Non-IEEE Mode

The floating-point Non-IEEE mode option defined by the PowerPC Architecture is implemented, but it is for hardware debug purpose only and not available in normal operations of the PPE.
Attempts to set the FPSCR[NI] bit to ‘1’ are ignored by the processor.

11.5. Nop Forms of the OR/ORI Instructions

Book I: PowerPC User Instruction Set Architecture identifies the preferred "**nop**" instruction (an instruction that does nothing) as **ori 0, 0, 0**. This implementation has no performance preference for one form of **nop** over any other. For example, **or 0, 0, 0** has the same performance as **ori 0, 0, 0**. This is because the processor does not remove **nop** instructions from the pipeline after they are issued. This means that the FXU executes the **nop** instruction even though it has no affect on processor state. This form of the **nop** is still recommended for this implementation as it is the most portable option from a programmer's point of view.

This implementation supports a subset of the program priorities defined in OR instruction section (*Section 3.4.1)* of *Book III: PowerPC Operating Environment Architecture*. The implementation also adds specific forms of the **nop** instruction for performance enhancements. *Table 11-1* below lists the supported priorities. For details on the behavior of the processor in response to different program priorities, see *section 6.6.3.2 “Thread Priority”* on page 50. For details on the behavior of the processor in response to the stalling forms of the **nop** instruction, see *“Dispatch Stalling Nop Instructions”* on page 48.

Programming Note:

This implementation does not support the “very low”, “medium low”, “medium high”, and “very high” priority definitions (Rx = 31, 6, 5, and 7 respectively). The **nop**s indicating a “medium low”, “medium high”, and “very high” priorities (Rx = 6, 5, and 7) have no affect on the processor (that is, they are ignored). The **nop** indicating a “very low” priority (Rx = 31) is used for a different purpose on this implementation that is inconsistent with the architecture. Software should take special note that code written to use the “very low” priority defined in the architecture may perform poorly on this implementation, but should not cause any incorrect results as a consequence of execution. Similarly code written expecting the behavior defined by this implementation may perform poorly on other implementations but should not cause incorrect results.

Table 11-1. Special nop forms

OR Form	Extended Mnemonic	Description
ori 0, 0, 0	**nop**	Preferred form for nop instruction.
or 1, 1, 1	**cctpl**	Change current thread priority to low.
or 2, 2, 2	**cctpm**	Change current thread priority to medium.
or 3, 3, 3	**cctph**	Change current thread priority to high.
or 28, 28, 28	**db8cyc**	Cause the current thread to block at dispatch for 8 cycles. This is used for multi-thread compiler optimizations of long-latency VSU dependencies
or 29, 29, 29	**db10cyc**	Cause the current thread to block at dispatch for 10 cycles. This is used for multi-thread compiler optimizations of long-latency VSU dependencies
or 30, 30, 30	**db12cyc**	Cause the current thread to block at dispatch for 12 cycles. This is used for multi-thread compiler optimizations of long-latency VSU dependencies
or 31, 31, 31	**db16cyc**	Cause the current thread to block at dispatch for 16 cycles. This is used for multi-thread compiler optimizations of long-latency VSU dependencies
ori 1-31, 1-31, 0 **or 0, 0, 0** **or 4-27, 4-27, 4-27**	**N/A**	Nonspecial forms for nop instruction

11.6. Storage Model

11.6.1. Storage Access Ordering

The architecture defines a weakly ordered storage model for most types of storage access. Certain out-of-order bus transactions are allowed in this model. As a result, if strongly ordered storage accesses are required, software must use the appropriate synchronizing instruction (**sync**, **eieio**, or **lwsync**) to enforce order explicitly, or perform these accesses to regions marked with attributes that require the hardware to enforce strong ordering.

11.7. Storage Control Instructions

This section gives implementation details for instructions that control storage.

11.7.1. Instruction Cache Block Invalidate (icbi)

The instruction cache block size for **icbi** is 128 bytes. The L1 ICache does not support a snoop operation. Hence, the L1 ICache and the L1 DCache are not necessarily consistent with modifications to those storage locations. This in turn requires that the system data storage error handler be invoked if an address translation or storage protection violation occurs.

Programming Note:

The effective address specified by (RA | 0) + (RB) of the **icbi** instruction is translated as a data address by the Load Store Unit (LSU). Hence, data address translation rules are used, and any exception that results is handled as a Data Storage exception.

11.7.2. Data Cache Block Touch (dcbt and dcbtst)

The data cache block size for **dcbt** and **dcbtst** is 128 bytes.

The **dcbtst** instruction has the same behavior as the TH = ‘0000’ form of the **dcbt** instruction as described below.

These instructions act like a load instruction from the view of the cache hierarchy and the TLB with the following exceptions. If data translation is enabled (MSR[DR] = '1'), and a segmentation or page fault results then the instruction is cancelled and no exception occurs. If a TLB miss results and LPCR[TL] = '0' (hardware tablewalk mode), the TLB is reloaded and the corresponding reference bit is set if a matching page table entry is found. If the page-protection attributes prohibit access, the page is marked caching-inhibited (I = '1'), LPCR[TL] = '1' (software tablewalk mode), or the page is marked guarded (G = '1'), then the instruction is cancelled and does not reload the cache. Otherwise, the instruction checks the state of the L1 DCache. If the data requested is not present, then a reload request is sent to the L2. If the data is not present in the L2, then the L2 requests the data from memory and only the L2 is reloaded (the data is not forwarded to the L1). However, if TH = ‘0000’, the L1 is reloaded as well. Similarly, if the data is in the L2 already, then data is not forwarded to the L1. However, if TH = ‘0000’, the L1 is reloaded as well.

If the **dcbt** or **dcbtst** instruction reloads a cache block, the replacement attributes for the cache (e.g. the least recently used information) are updated.

This processor implements the optional form of the **dcbt** instruction defined in *Section 5.2.1.1 of Book II: PowerPC Virtual Environment Architecture, Version 2.02*. The **dcbt** instruction is used to setup a data stream. Each data stream is managed independently.

Excerpt of the instruction description from the Book II is shown below for convenience.

Data Cache Block Touch X-form

dcbt RA, RB, TH

31	/	TH	RA	RB	278	/
0	6	7	11	16	21	31

Let the effective address (EA) be the sum (RA|0)+(RB).

TH= ‘0000’

The dcbt instruction provides a hint that the program will probably soon load from the block containing the byte addressed by EA.

TH=‘1000’

The dcbt instruction provides a hint that describes certain attributes of a data stream,
and optionally indicates (by the UG bit) that the program will probably soon load from the stream.
The EA is interpreted as follows.

EATRUNC	D	UG	/	ID
0	57	58	59	60 63

bit(s)		
0:56	EATRUNC	High-order 57 bits of EA of first unit of data stream
57	Direction (D)	0: ascending, 1: decending
58	Unlimited/GO (UG)	0: no info, 1: unlimited number of units, hint to start stream
60:63	Stream ID (ID)	

TH=‘1010’

The dcbt instruction provides a hint that describes certain attributes of a data stream,
or indicates that the program will probably soon load from data streams that have been described using dcbt instructions in which TH(0) = ‘1’
or will probably no longer load from such data streams.
The EA is interpreted as follows.

///	GO	S	///	UNITCNT	T	U	/	ID
0	32	33 34	35	47 56	57	58	59	60 63

bit(s)		
32	GO	1: hint to start stream
33:34	Stop (S)	‘10’: hint to stop specified stream ID
		‘11’: hint to stop all stream IDs
47:56	UNITCNT	the number of units
57	Transient (T)	
58	Unlimited/GO (UG)	1: unlimited number of units
60:63	Stream ID (ID)	

In the PPE implementation, the Touch Hint (TH) field of the instruction has the following implementation-specific characteristics:

- The following forms of the instruction are implemented: TH = ‘0000’, ‘1000’, and ‘1010’. All other TH(0) = ‘0’ forms are not supported on this processor and are implemented the same as TH = ‘0000’. All other TH(0) = ‘1’ forms of the instruction are not supported on this processor and are ignored.
- For the TH = ‘1010’ form of the instruction, the Transient field (bit 57 of the effective address) is ignored by the implementation.

Stopping a stream terminates the stream and relinquishes its Data Prefetch Engine (DPFE) queue entry. The following conditions can terminate a stream:

- When software issues an all stop command (TH = '1010', GO = '0', Stop = '11'), all active streams for the issuing thread are terminated.
- When software issues a single stop command (TH = '1010', GO = '0', Stop = '10') to a specific stream, the specified stream is terminated.
- When software issues a **dcbt** instruction with either TH = '1000' or TH = '1010' that has the same Stream ID as an already active stream, the active stream is terminated and restarted with the state indicated by the new instruction.
- When a stream setup with TH = '1010' reaches the UNITCNT it is terminated.
- When a stream reaches its page boundary, the stream is terminated. For pages larger than 64 KB, the stream is nonetheless terminated at each 64 KB boundary.
- When a thread enters real mode operation (that is, set MSR[DR] to '0'), all the streams for the thread are terminated.
- When a thread is no longer enabled, all the streams for the thread are terminated.
- A **tlbie** instruction or a snooped TLBIE bus command terminates all the streams for both threads.

There are a total of 8 streams available. These are split into four streams for each thread.

An MMIO register in the CIU (CIU_ModeSetup) alters the behavior of the **dcbt** instruction. See *Cell Broadband Engine™ Registers* for details.

11.7.3. Data Cache Block Zero (dcbz)

The data cache block size for **dcbz** is 128 bytes.

The function of **dcbz** is performed in the L2 cache. As a result, if the block addressed by the **dcbz** is present in the L1 DCache, the block is invalidated before the operation is sent to the L2 cache. The L2 cache gains exclusive access to the block (without actually reading the old data), and performs the zeroing function in a broadside manner. That is, all bits in the cache line are set to zero, which means the data cannot be discovered by another process. The **dcbz** is handled like a store and is performed in the appropriate storage model order. It does not bypass any of the store queues. In general, the performance of **dcbz** is better than a sequence of 8 Byte stores, since it zeros 128 Bytes at a time.

Regardless of whether or not the cache block specified by the **dcbz** instruction contains an error (even one that is not correctable with ECC), the contents of the block are set to zero in the L2 Cache.

If the block addressed by the **dcbz** instruction is in a memory region marked caching-inhibited (I = '1'), or if the L1 DCache is disabled (via HID4[en_dcway] = '0000'), then the instruction causes an alignment interrupt.

11.7.4. Data Cache Block Flush (dcbf) and Data Cache Block Store (dcbst)

The data cache block size for **dcbf** and **dcbst** is 128 bytes.

A **dcbf** instruction causes all caches for all processors to write modified contents back to memory and invalidate the data in the cache for the block containing the byte addressed by the instruction. A **dcbst** behaves the same except that the data in the cache remains valid (if present).

The **dcbf** and **dcbst** instructions will probably complete before the operation they cause (flush or write-back of the cache) has completed. A context-synchronizing instruction (CSI) ensures that the effects of these instructions are complete for the processor issuing the CSI.

11.7.5. Load and Reserve and Store Conditional Instructions

The coherency granule size in the PPU is 128 bytes.

There is one reservation register per thread. The following events affect the state of the reservation register:

- Local execution of a **lwarx** or **ldarx** instruction (a new reservation is set for the thread executing the instruction)
- Local execution of a **stwcx.** or **stdcx.** instruction (the reservation is cleared for the thread executing the instruction regardless of the success of the operation or address match to the reservation register)
- Snooping of a store, **dcbz**, **dcbtst**, **dcbst**, or **dcbf** instruction from another processor that matches the reservation address for either thread (both threads are tested for a match; if a match occurs, the reservation is cleared)

An attempt to execute a non-word-aligned **lwarx** or **stwcx.**, or a non-doubleword-aligned **ldarx** or **stdcx.** causes an alignment interrupt. Similarly, an attempt to execute a **lwarx**, **ldarx**, **stwcx.**, or a **stdcx.** instruction to storage marked caching-inhibited causes a alignment interrupt.

11.7.6. Memory Barrier Instructions (sync, isync, and eieio)

The processor supports all types (L = 0, 1, and 2) of the **sync** instruction (heavyweight, lightweight, and pte). A sync with L = 3, which is reserved according to the architecture, and the heavyweight (L = 0) **sync** are treated the same as a pte (L = 2) **sync**. The heavyweight (L = 0) and **pte** (L = 2) **sync** instructions behave exactly the same on this implementation. This behavior covers the requirements of both the heavyweight and pte **sync** varieties as defined in the architecture.

The lightweight **sync** works only on the processor issuing it and is not sent to other processors in the system. The heavyweight and pte **sync** are broadcast to all processors in the system. Only processors operating in the same partition (as set in the LPIDR register) accept the broadcast **sync**; others ignore it.

The **isync** and **sync** instructions on this processor implementation wait to complete until all storage accesses associated with preceding instructions have been performed.

In the PowerPC Processor Storage Subsystem (PPSS), some store queues attempt to gather sequentially both non-caching-inhibited (cacheable) and caching-inhibited store operations to improve bandwidth. To avoid this behavior, software must insert either an **eieio** for guarded storage or a **sync** (L = 0, 1, or 2) for non-guarded storage to prevent gathering. The **eieio** instruction generally has better performance but can only be used to prevent gathering on guarded (G = '1') storage accesses.

The **eieio** transaction is broadcast onto the Element Interconnect Bus (EIB) to allow ordering to be properly enforced on the bus and in the memory system and processor I/O.

An **eieio** instruction issued on one thread does not have any effect on the other thread. However, there is a performance effect because all load or store operations (cacheable or noncacheable) on the other thread are serialized behind the **eieio** instruction.

12. Extended Architecture Compliance

This chapter provides details on extended architecture implemented by this processor that is not part of PowerPC Architecture.

12.1. Vector/SIMD Multimedia Extension Processor

This processor implements a Vector/SIMD Multimedia Extension unit (VXU) as part of the Vector/Scalar unit (VSU). The architecture is defined in *PowerPC Microprocessor Family: Vector/SIMD Multimedia Extension Technology Programming Environments Manual* Version 2.06c. This section details unique features of this processor implementation of the Vector/SIMD Multimedia Extension specification. If an implementation feature matches the Vector/SIMD Multimedia Extension specification without need for further elaboration, then the feature is not presented in this section (to avoid redundancy with the specification).

12.1.1. Java Compliant Mode Execution

When VSCR[NJ] = '0', the Vector/SIMD Multimedia Extension processor is configured to operate in Java Compliant Mode. This mode is consistent with IEEE standards for binary floating-point arithmetic. Operating in this mode is likely to reduce performance of Vector/SIMD Multimedia Extension instructions that have any of the following (due to internal flushing):

- denormalized operands
- denormalized results

When in non-Java mode (VSCR[NJ] = '1'), denormalized operands and results as well as underflow results are treated as zeros.

12.1.2. VXU Graphics Rounding Mode

Graphics rounding mode can be used to improve performance on graphic-oriented workloads. When HID1[grap_mode] is set to ‘1’ and VSCR[NJ] is set to ‘1’, the VXU operates in graphics rounding mode. (HID1[grap_mode] = ‘1’ with VSCR[NJ] = ‘0’ is an invalid configuration, and the results are undefined.)

The following set of rules apply to graphics rounding mode:

- The VXU unit operates in non-Java mode.
- Denormal inputs are flushed to zero (same as non-Java mode behavior).
- Except where noted the rounding mode is round to zero.
- Infinity and NaN inputs are operated upon as normal numbers, with the positive overflow boundary moved from x'7F80_0000' to x'7FFF_FFFF' and the negative overflow boundary moved from x'FF80_0000' to x'FFFF_FFFF'.
- If the result is an underflow the result will be zero (same as non-Java mode behavior).
- If the infinitely precise unrounded result would denormal, the result will be zero (same as non-Java mode behavior).
- If the result is greater than the new positive_overflow_boundary, then the result will be x’7FFF_FFFF'.
- If the result is less than the new negative_overflow_boundary, then the result will be x’FFFF_FFFF'.

The following subsections detail graphics rounding mode behavior for individual instruction classes.

Math instructions (Multiply-Add, Multiply-Subtract, Add, Subtract)

Since there are no infinity or NaN inputs, there will not be a NaN result for infinity – infinity or infinity × 0. Infinity or NaN multiplied by zero will have a result of zero.

Floating-Point Compares

NaN and Infinity inputs are treated as large contiguous numbers, and they are correctly compared. The cases for comparing +0 to –0 are the same as defined for non-graphics rounding mode.

Reciprocal Estimate

The same results as non-Java mode are generated except that NaN and Infinity inputs are interpreted as large numbers and translated to correct mathematical estimates.

Reciprocal Square Root Estimate

The instruction takes the absolute value of the input, then return a result that is equivalent to the non-Java mode reciprocal square root estimate of the positive value. NaN and Infinity inputs are interpreted as large numbers and translated to the correct mathematical estimate.

Logarithm Base 2 Estimate

The same results as non-Java mode are generated except that NAN and Infinity inputs are interpreted as large numbers and translated to correct mathematical estimates. In non-Java mode the log of a negative number is a NaN result. For negative inputs in graphics rounding mode, the result is the log of the absolute value of the input, except when the input is +0 or –0. If the input is +0, the output is x’FFFF_FFFF'. If the input is –0, the output is also x’FFFF_FFFF'.

Power of 2 Estimate

The same results as non-Java mode are generated except that NaN and Infinity inputs are interpreted as large numbers and translated to correct mathematical estimates. The detection for an overflow boundary is the same as for non-Java mode. However, the result is an overflow boundary for graphics rounding mode (given above).

Scaled Convert to integer

Since the non-Java rounding mode is truncated for these instructions, the result is the same as non-Java mode except for NaN and Infinity inputs. NaN and Infinity inputs are too large for the integer range, so they cause the same integer saturation results as non-Java mode for Infinity inputs. Positive NaN inputs give the same result as positive infinity in non-Java mode. Negative NaN inputs give the same result as negative infinity in non-Java mode.

Scaled Convert from integer

The results are the same as non-Java mode. The rounding mode is round-to-nearest-even. Since the input is an integer, there are no differences for NaN and infinity inputs.

Round to Floating-Point integer

Since the rounding mode is part of the instruction, the rounding mode is not suppressed as in the default graphic mode rules. The result is the same as for non-Java mode. There is a form of the instruction that uses the rounding mode truncate.

12.1.3. Vector Multimedia Registers

- VRSAVE Register—The VRSAVE register is a 32-bit register maintained and managed only by software. It is intended to maintain a record of dead (not used in the current process) or live (used in the current process) registers. Each of the 32 bits represents one of the architected vector registers.
- Vector Status and Control Register (VSCR)—The VSCR is a special 32-bit register that is read and written in a manner similar to the FPSCR in the scalar floating-point unit defined in the PowerPC architecture. Special instructions (**mfvscr** and **mtvscr**) are provided to move the VSCR from and to a vector register. When moved to or from a vector register, the 32-bit VSCR is right-justified in the 128-bit vector register. When moved to a vector register, bits 0 – 95 of the vector register are set to zeros.

  Note: The VSCR is not a Special Purpose Register (SPR).
- Vector Registers (VR)—There are 32 vector registers for carrying out VXU operations.

12.1.4. Accuracy of VXU Floating-Point Instructions

vrefp

Table 12-1. Operands with Results for vrefp

Sign	Exponent (unbiased)	Significand	**vrefp** result
±	Zero	Don't care	±Infinity
±	Exponent < 126	Don't care	±(Normal | Infinity)
±	Exponent = 126	1.fraction = 1.0	±Normal
±	Exponent = 126	1.fraction ≠ 1.0	Denormal: Internal Flush
±	Exponent = 127	Don't care	Denormal: Internal Flush
±	Infinity	Don't care	±Zero

vexptefp

If the operand of a **vexptefp** instruction is such that the fully accurate result would be denormal, the hardware returns a zero result, as described in the table below.

Table 12-2. Operands with Results for vexptefp

Sign	Exponent (unbiased)	Significand (binary)	Operand value	**vexptefp** result
0 (+)	Exponent ≥ 7	Don't care	Operand ≥ 128	+Infinity
0 (+)	Exponent ≤ 6	Don't care	0 ≤ Operand < 128	+Normal
1 (–)	Exponent ≤ 5	Don't care	–64 < Operand ≤ 0	+Normal
1 (–)	Exponent = 6	1.fraction ≤ 1.1111100	–126 ≤ Operand ≤ –64	+Normal
1 (–)	Exponent = 6	1.fraction > 1.1111100	–128 < Operand < –126	+Zero
1 (–)	Exponent = 7	1.fraction ≤ 1.0010101	–149 ≤ Operand ≤ –128	+Zero
1 (–)	Exponent = 7	1.fraction > 1.0010101	–256 < Operand < –149	+Zero
1 (–)	Exponent > 7	Don't care	Operand ≤ –256	+Zero

Accuracy of vexptefp and vlogefp

The **vexptefp** and **vlogefp** instructions offer significantly higher accuracy and continuity characteristics than the Vector/SIMD Multimedia Extension architecture specification requires and what previous implementations have delivered. Previous implementations of **vexptefp** and **vlogefp** went unused by software developers due to discontinuities present in those implementations.

Specular highlighting in real-time 3-dimensional graphics is one application that typically uses the **vexptefp** and **vlogefp** instructions. Specular highlights in 3D graphics simulate the reflection of a light source on a surface without using more calculation-intensive operations such as ray tracing. The rate at which the highlight fades from its lightest color in the center, to the darkest outer portion is calculated by the following equation:

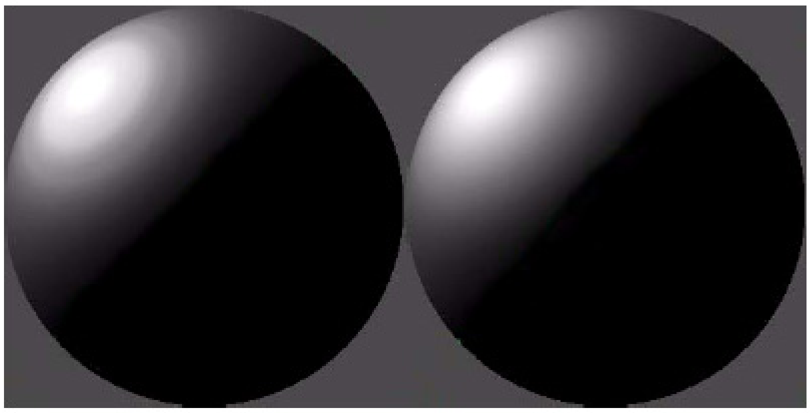
$$\text{intensity} = 2^{k \times \log_2(\text{Cos}(\Theta))}$$

where k is a property of the surface (a higher number being a smoother surface) and theta (Θ) is the angle between the ray connecting the surface with the eyepoint, and the ray representing the reflection off the surface from the light source.

Continuity Illustrations for vexptefp and vlogefp

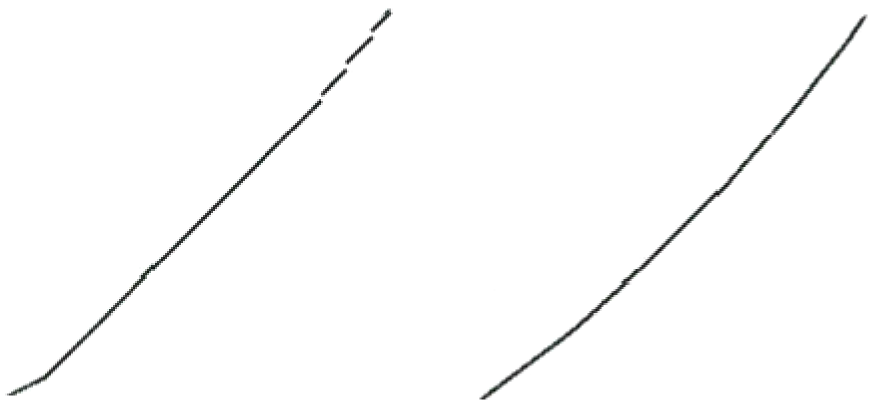
Figure 12-1 on page 119 shows two spheres rendered using two different implementations of **vexptefp** and **vlogefp**. The sphere on the left is rendered with a previous implementation, while the sphere on the right uses the new improved implementation. The sphere on the left has visible banding where the intensity gradient of the sphere's specular highlight is not continuous. The specular highlight of the sphere on the right shows a more continuous gradient. This improvement allows for much greater realism in computer graphics applications.

Figure 12-1. Comparison of Floating-Point Estimate Implementations



The discontinuities in the prior implementation are visible in the leftmost image of *Figure 12-2*, where a range of outputs from the **vexptefp** instruction has been plotted verses the input. The rightmost image of *Figure 12-2* shows the same range produced with the new improved **vexptefp** implementation, where any discontinuities are no longer visible.

Figure 12-2. Prior Implementation (left) and Improved Implementation (right) of vexptefp



Accuracy of vexptefp and vlogefp

While the new implementation shows improved continuity, both **vexptefp** and **vlogefp** have greatly improved accuracy as well. *Table 12-3* compares the relative and worst-case error between a prior implementation and the improved implementation. The accuracy is at least 66 percent better for each instruction with the new implementation.

Table 12-3. Accuracy Comparisons of vexptefp and vlogeflp

Error	Prior Implementation	New Implementation
vexptefp Absolute Error	0.896×2^{-5}	0.302×2^{-5} (0.604×2^{-6})
vexptefp Relative Error	0.822×2^{-5}	0.157×2^{-5} (0.628×2^{-7})
vlogefp Absolute Error	0.785×2^{-5}	0.201×2^{-5} (0.804×2^{-7})
vlogefp Relative Error (operand [VB] > 1.125)	0.479×2^{0}	0.071×2^{0} (0.568×2^{-3})

12.2. Cache Replacement Policy Management

This processor provides a method of controlling the L2 and TLB cache replacement policy based on a class identifier (classID). The architecture for this method is defined in the *"Cache Replacement Management Facility"* section of the *Cell Broadband Engine™ Architecture*.

The classID is generated from the effective address (EA) specified by instruction fetches or by data access from load or store instructions. Instruction fetches include fetches for sequential execution, speculative and non-speculative branch targets, prefetches, and interrupts. Cache management instructions are treated like load or store instructions with respect to classID. The classID is used to generate an index to a L2 or TLB Replacement Management Table (RMT) that privileged software can configure to control the replacement policy.

12.2.1. Address Range Registers

A set of Address Range Registers are implemented by the processor for mapping an instruction or data effective address to a classID. This classID is used as an index for both the TLB_RMT and L2 RMT.

An Address Range is a naturally-aligned range that is a power of 2 in size and is between 4 KB and 4 GB, inclusive. An Address Range is defined by two types of registers; a Range Start Register (RSR) and a Range Mask Register (RMR). For each Address Range, there is an associated ClassID Register (CIDR) that specifies the classID. The RSR, RMR, and CIDR are generic labels for a group of registers. As depicted in *Table 12-4*, for each processor thread there are two sets of RSRs, RMRs, and CIDRs for load/store data accesses and two sets for instruction fetches. The Address Range Registers are accessible by the PowerPC Architecture Move To/From Special Purpose Register instructions. Access to an RSR, RMR or CIDR is only allowed in privileged state. Refer to *Data Address Range SPRs* and *Instruction Range SPRs* in *Appendix: SPR Definitions*.

Table 12-4. Address Range Registers for each Processor Thread

type of operation	Address Range		associated classID
	RSR *	RMR *	CIDR *
data access	Data Address Range Start Register 0 (DRSR0)	Data Address Mask Register 0 (DRMR0)	Data Class ID Register 0 (DCIDR0)
	Data Address Range Start Register 1 (DRSR1)	Data Address Mask Register 1 (DRMR1)	Data Class ID Register 1 (DCIDR1)
instruction fetch	Instruction Range Start Register 0 (IRSR0)	Instruction Range Mask Register 0 (IRMR0)	Instruction Class ID Register 0 (ICIDR0)
	Instruction Range Start Register 1 (IRSR1)	Instruction Range Mask Register 1 (IRMR1)	Instruction Class ID Register 1 (ICIDR1)
* generic label for 4 registers listed below in the column			

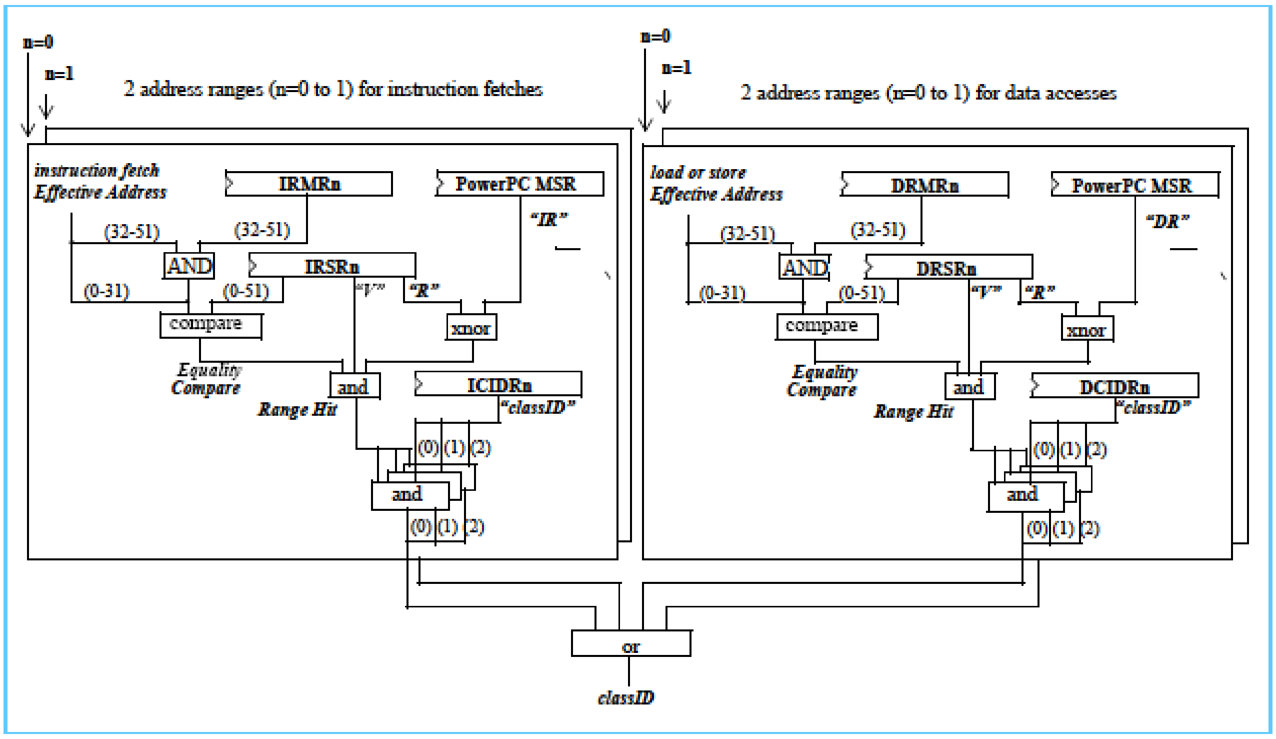
If all the following conditions are met, the particular Address Range defined by an RSR and RMR pair applies to a given EA and a "range hit" is said to have occurred:

- RSR[63] = '1'
- RSR[0:51] = EA[0:31] | | (EA[32:51] & RMR[32:51])
- If the operation is a load or store, then
  - RSR[62] = MSR[DR] and
  - the RSR and RMR pair used in the above conditions must be DRSR0 and DRMR0 or DRSR1 and DRMR1
- else (the operation is an instruction fetch)
  - RSR[62] = MSR[IR] and
  - the RSR and RMR pair used in the above conditions must be IRSR0 and IRMR0 or IRSR1 and IRMR1

If there is no range hit for a given effective address, the classID has a value of zero.

In effect, RMR defines the size of the range by selecting the bits of an EA used to compare with the RSR. The upper bits of an RSR contain the starting address of the range and the lower bits contain a relocation mode (Real or Virtual) and a valid bit. The size of the range must be a power of two. The starting address of the range must be a range size boundary.

Figure 12-3. Generation of ClassID from the Address Range Registers for each Processor Thread



Programming Note:

To avoid confusion about the classID value, it is recommended that software ensure that data address ranges 0 and 1 do not overlap such that both have a simultaneous range hit. Likewise, it is recommended that software ensure that instruction address ranges 0 and 1 do not overlap such that both have a simultaneous range hit.

12.2.2. L2 Replacement Management Table (RMT)

For each classID, the L2 replacement management table (RMT) defines which ways of the L2 cache are eligible to be replaced when an L2 Cache miss occurs.

The L2 RMT can be used by software to achieve various results. Some examples of this are:

- Lock an address range into the cache so that an access to the block always get an L2 cache hit. The address range is "locked" into the L2 cache by first making it valid in the L2 cache and then configuring the L2 RMT to prevent the block from being displaced from L2 cache.
- Limit an address ranges to one or more ways of the L2 cache without locking these locations and without prohibiting accesses for other data from replacing for those ways. This is useful for some kinds of accesses over large data structures in order to prevent such accesses from flushing a large portion of the L2 cache.
- Allow an application to only use a limited number of ways of the L2 cache while reserving other ways for other applications. This can be useful to ensure a certain performance level for applications, and that performance is dependent on L2 cache miss rate. Limiting an application to a subset of the L2 cache prevents the application from displacing all data cached by another application. Such data is not necessarily locked into the L2 cache, but can simply be the latest set of data used by the other application.

The L2 cache replacement policy is controlled by privileged software through a Replacement Management Table (RMT) when the L2 cache is not in direct mapped mode (replacement management is not supported in direct mapped mode). The processor supports an 8-entry RMT for the eight-way set associative L2 cache. The 3-bit classID is used as an index into the RMT to select one of the RMT entries. Each entry of the RMT contains a replacement enable bit for each way of the cache. The RMT is instantiated in the L2 RMT Data Register. The 3-bit classID selects one of the eight fields in this register. *Figure 12-4* depicts the L2 RMT Data Register. The RMT function can be enabled or disabled by L2 Mode Setup Control Register [62]. If disabled, all ways of the L2 are eligible to be replaced when any L2 cache miss occurs, and the way to be replaced is selected via an LRU algorithm.

Figure 12-4. L2 RMT Data Register

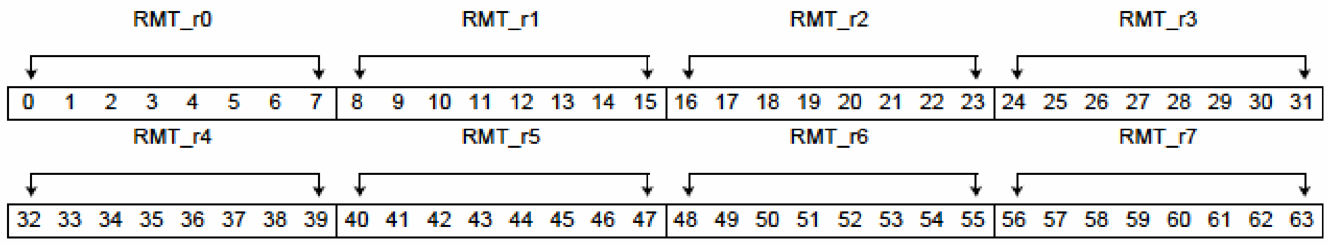


Figure 12-4 indicates which ways in the set associative cache can be replaced when there is an L2 cache miss for the respective classID. Each RMT entry has a bit for each way of the 8-way set associative L2 cache. If the RMT entry bit for the respective way is a '0', that way is not replaced by that corresponding class. If the RMT entry bit for the respective way is a '1', that way is eligible to be replaced by that corresponding class. If more than one way is eligible to be replaced for a particular classID, a binary-tree pseudo LRU algorithm is used to select which one of the eligible ways will be replaced.

If all eight bits of the indexed RMT entry are zero, then the RMT entry is treated as if it were x'FF'. In this case, all L2 cache ways are eligible to be replaced.

12.2.3. TLB Replacement Management

The processor provides a method of controlling the TLB replacement policy based on a class identifier (classID). Software can use the PPE_TLB_RMT table to lock translation entries into the TLB by reserving a particular way of the TLB to a specific program effective-address range. See *Section 12.2.1 "Address Range Registers"* on page 121 for details of on how effective addresses are mapped to classIDs. The classID is used to generate an index to the PPE TLB Replacement Management Table (PPE_TLB_RMT) that privileged software can configure to control the replacement policy. For each classID, the PPE_TLB_RMT field defines which ways of the TLB are eligible to be replaced when a TLB miss occurs.

Programming Note:

TLB replacement management is only for "hardware tablewalk" mode (LPCR[TL] = '0'). When in "software tablewalk" mode (LPCR[TL] = '1'), software controls which entry of the TLB is replaced directly.

A single 64-bit PPE_TLB_RMT SPR register is provided to implement this mechanism (see *Section PPE Translation-Lookaside Buffer RMT Register* (PPE_TLB_RMT) on page 141). The high-order 32-bits of this register are non-implemented and reserved for future use. The low-order 32-bits are divided into eight four-bit RMT fields, where each bit of the RMT represents a way of the cache. Each four-bit field corresponds to a classID.

RMT0 is the default entry and should be used by software to specify the replacement policy for any effective address range that is not mapped by the Address Range Registers (see *section 12.2.1 "Address Range Registers"* on page 121).

Each bit of an RMT field has a one-to-one correspondence to each of the four ways in the TLB. For each bit with a value of 1, the tablewalk algorithm can replace an entry in this way should a TLB miss occur in the effective address range that maps to the ClassID for this RMT field. If multiple bits are set within an RMT field, a pseudo

LRU (binary-tree algorithm) chooses between the valid replacement ways. For example, a translation with ClassID 5 is requested. ClassID 5 corresponds to RMT5, which has a value of ‘1001’. If a TLB miss occurs, TLB ways 0 and 3 are valid candidates for replacement (pseudo-LRU chooses way 0 initially and then points to way 3 when another miss occurs).

If all bits in an RMT field are set to 0 (‘0000’), then the hardware treats this RMT field as if it had been set to ‘1111’, therefore allowing replacement to any way in the TLB.

The replacement policy used in the TLB is a 3-bit pseudo-LRU policy in binary tree format. It has 256 entries, each representing a single congruence class in the TLB. The LRU uses the following basic rules:

1. The LRU is updated every time an entry in the TLB is hit by a translation operation. The entry that was referenced becomes the most recently used and the LRU bits are updated accordingly. However, the LRU is not updated when software reads the TLB for debug purposes via **mfspr** PPE_TLB_VPN or **mfspr** PPE_TLB_RPN.
2. On a TLB miss, the update algorithm determines which entry to replace using the following criteria:
    a. Any invalid entry is replaced. If more than one entry is invalid then the leftmost (or lowest numbered) way is chosen as invalid.
    b. The entry that the RMT determines is the least recently used by the LRU and is eligible for replacement is replaced.

12.3. TLB Management Instructions and ERAT Coherency

This facility implements the architecture specified in the "*Translation Lookaside Buffer Management*" section of the *Cell Broadband Engine™ Architecture*. This processor allows for software management of the TLB by the hypevisor when LPCR[TL] = '1'.

Note: Software management of the TLB is restricted to the hypervisor (since memory is a system resource). Any attempt to execute TLB management instructions when MSR[PR, HV] is not equal to '01' causes an Illegal Instruction type of Program interrupt. Reading of the TLB is strictly for hardware debugging purposes only.

All TLB misses result in a page fault leading to a Storage interrupt. Software can then write the new entry to the TLB as described below. Additionally, software can use this facility to preload the TLB with translation entries that are anticipated in future code execution. This is useful for avoiding TLB misses altogether.

The TLB is mapped into SPRs for the purpose of software management, as shown in *Table 12-5*. See *Table 5-1 PowerPC Processor Element SPRs* on page 14 for address and privileged and synchronization information for reading and writing these registers. *Table 5-1* also provides implementation details for the fields within these registers.

The PPE Translation-Lookaside Buffer Index Hint Register (PPE_TLB_Index_Hint),
PPE Translation-Lookaside Buffer Index Register (PPE_TLB_Index),
PPE Translation-Lookaside Buffer Virtual-Page Number Register (PPE_TLB_VPN), and
PPE Translation-Lookaside Buffer Real-Page Number Register (PPE_TLB_RPN)
defined in the *Cell Broadband Engine™ Architecture* are implemented.
The TLB_Invalidate_Entry and TLB_Invalidate_All registers are not implemented.

Software should use the PowerPC Architecture **tlbie** instruction for invalidating entries in the TLB. In *Table 12-5* the command names in parentheses are recommended compiler mnemonics.

Table 12-5. TLB Software Management

TLB Operation	Code Sequence

Write an entry (**tlbmte**)	**mtspr** PPE_TLB_Index, RB;
	mtspr PPE_TLB_RPN, RB;
	mtspr PPE_TLB_VPN, RB;

Read from Index Hint (**tlbmfih**)	**mfspr** RT, PPE_TLB_Index_Hint;
Set the index for writing (**tlbmti**)	**mtspr** PPE_TLB_INDEX, RB;

Debug: Read from entry (**tlbmfe**)	**mtspr** PPE_TLB_INDEX, RB;
	mfspr RT1, PPE_TLB_VPN;
	mfspr RT2, PPE_TLB_RPN;

Invalidate an entry	**tlbie**(l)
Invalidate all entries	See *Section 8.4.1 TLB Invalidation* on page 72

Writing the PPE_TLB_VPN register causes an atomic write of the TLB with the last written contents of PPE_TLB_RPN and the Lower Virtual Page Number (LVPN) field of PPE_TLB_Index. The entry in the TLB that is written is the entry pointed to the last-written value of PPE_TLB_Index.

Programming Note:

- The TLB tags each entry with the processor LPID value at the time the entry is written. The hypervisor should make sure to set the Logical Partition ID (LPID) register to the value it wishes to have the TLB entry tagged with at the time the entry is written.
- Software must set bit 56 of the PPE_TLB_VPN register (the low-order bit of the Abbreviated Virtual Page Number [AVPN] field) to 0 when writing a 16-MB TLB entry.

Reading of the PPE_TLB_VPN or PPE_TLB_RPN registers returns the Page Table entry data associated with the TLB entry pointed to by the last-written value of PPE_TLB_Index. Reading the PPE_TLB_Index_Hint register returns the location that the hardware would have chosen to replace when the last TLB miss occurred as a result of a translation for the thread performing the read operation (PPE_TLB_Index_Hint is duplicated per thread). If no miss has occurred, the register returns its POR value (typically all zeros).

Programming Note:

- Reading of the TLB is provided for hypervisor software debugging purposes only. The LPID and LVPN fields of the TLB entry are not accessible to the hypervisor (since they are not defined in the page table entry which the PPE_TLB_VPN and PPE_TLB_RPN registers reflect). This makes it impossible to save and restore the TLB directly.
- Bit 56 of the PPE_TLB_VPN register (the low-order bit of the AVPN field) is undefined when the TLB entry is a 16-MB page.

When reading or writing the TLB software must follow these rules:

1. Both threads should not attempt to write the TLB at the same time. Doing so can lead to boundedly undefined results because there is a non-atomic sequence of register updates required for TLB writes (even though the write to the TLB itself is atomic). For example, the following situation should be avoided: thread 0 writes the PPE_TLB_Index and PPE_TLB_RPN registers, then thread 1 writes over the PPE_TLB_Index register, then thread 0 writes the PPE_TLB_VPN register. The result is that the wrong TLB entry is written since thread 1 caused the PPE_TLB_Index register to point to the wrong location in the TLB. An easy way to solve this is for the hypervisor to reserve a lock when writing the TLB.

2 Because reading the TLB via mtspr and mfspr commands requires first writing the PPE_TLB_Index register, rule 1 applies. In other words, the same hypervisor lock used to write the TLB should also be acquired to read the TLB using the mtspr and mfspr commands. This prevents one thread from corrupting the PPE_TLB_Index pointer that the other thread is using to read the TLB. Failure to acquire a lock can lead to boundedly undefined results.
Also, because reading the TLB using the mtspr and mfspr commands is a non-atomic sequence, hardware updates of the TLB by the other thread should be prevented. For example, the following situation should be avoided: thread 0 reads the PPE_TLB_RPN register, then thread 1 performs a hardware tablewalk update of the same TLB entry, then thread 0 reads the PPE_TLB_VPN register. The PPE_TLB_RPN and PPE_TLB_VPN results do not correspond in this case. Because reading of the TLB using mtspr and mfspr commands is intended for debug use only, the hypervisor should stop the other thread from accessing the TLB while performing this operation.

3 There is no locking required to translate in the TLB. Because writes to the TLB itself are atomic, the thread translating in the TLB sees either the previous or the new TLB entry, but never an intermediate result. This is analogous to hardware updates of the TLB.

4 Software must at all times remember that the TLB is a non-coherent cache of the page table. If the TLB is completely software-managed, then it is software's responsibility to make sure that all TLB contents are maintained properly and that coherency with the page table is never violated.
Note: Invalidating a TLB entry requires use of the **tlbie** or **tlbiel** instruction. This is required so that the ERATs maintain coherency with the TLB. This means, for example, that simply writing the valid bit to '0' for a TLB entry is not sufficient for removing a translation. Only the **tlbie**(**l**) instruction properly back-invalidates the ERATs.

5 Software must maintain Reference and Change (R and C) bits in the page table. See Section *8.4.3.2 Reference and Change Bit Updates* on page 77 for R and C bit maintenance by hardware tablewalk. If software writes a TLB entry with C = '0', a subsequent store operation takes a page fault if in software tablewalk mode. The R bit is ignored by the TLB, so writing R = '0' will be ignored and treated like R = '1'.

Note: Failure of the hypervisor to properly synchronize TLB read and write operations can lead to undefined results throughout the system (requiring a system reset to recover).

Appendix: SPR Definitions

This section describes the implementation-specific Special Purpose Registers (SPRs) used in this implementation. For a complete listing of SPRs, see *Table 5-1 PowerPC Processor Element SPRs* on page 14. This table also contains architected SPRs that are not described in this document. Cross references to additional information about the architected SPRs are also provided in *Table 5-1*.

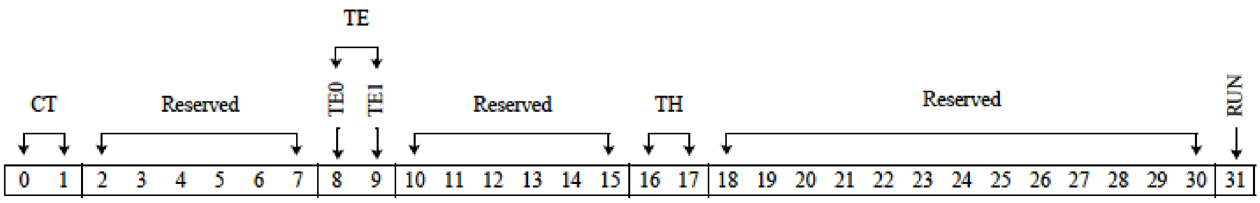
Notes:

The following information applies to the tables used at the beginning of each register description in this section:

1. In this section, power-on-reset (*POR*) is defined for this chip as the sequence that starts when power is first applied to the chip and ends when the load function completes.
2. *Value at Initial POR* is the value that was initialized during the scan initialization or configuration ring part of the POR sequence.
3. Some fields within the architected registers are not physically implemented. Writing to these fields has no effect, and reading from these fields returns 0. These fields are marked reserved. The rsvd_I bits are reserved and implemented. Writes to rsvd_I bits are preserved on a read.

Control Register (CTRL)

Register Short Name	CTRL	**Unit**	IU
Register Type	SPR Read/Write	**Width**	32 bits
Decimal SPR Number	136 (Read) 152 (Write)	**Access Type**	Read – Not privileged Write – Privileged
Value at Initial POR	All bits set to zero	**Set By**	Scan Initialization
Additional Information	PowerPC architected register with implementation-specific bits	**Reg. Duplicated for MT**	No



Bit(s)	Field Name	Description
0:1	CT	Current thread active (read only) These are read-only bits that contain the current thread bits for threads 0 and 1. Software can read these bit to determine which thread they are operating on. Only one current thread bit is set at a time. '00' Reserved '01' Thread 1 is reading CTRL '10' Thread 0 is reading CTRL '11' Reserved
2:7	Reserved	Bits are not implemented; all bits read back zero.
8:9	TE	Thread enable bits (read/write) The hypervisor state can suspend its own thread by setting the TE bit for its thread to '0'. The hypervisor state can resume the opposite thread by setting the TE bit for the opposite thread to '1'. The hypervisor state cannot suspend the opposite thread by setting the TE bit for the opposite thread to '0'. This setting is ignored and does not cause error. Thread enable bit for thread 0 (TE0) Thread enable bit for thread 1 (TE1) If thread 0 executes the **mtctrl** instruction: [TE0, TE1] Description '00' Disable/suspend thread 0, thread 1 unchanged '01' Disable/suspend thread 0, enable/resume thread 1 if it was disabled '10' Unchanged '11' Enable/resume thread 1 if it was disabled. If thread 1 executes the **mtctrl** instruction: [TE0, TE1] Description '00' Thread 0 unchanged, disable/suspend thread 1 '01' Unchanged '10' Enable/resume thread 0 if it was disabled, disabled/suspend thread 1 '11' Enable/resume thread 0 if it was disabled. **Note:** Software should not disable a thread when in trace mode (MSR[SE] or MSR[BE] set to '1'). Doing so will cause SRR0 to be undefined and can cause a system livelock hang condition.
10:15	Reserved	Bits are not implemented; all bits read back zero.
16:17	TH	Thread history If thread A writes CTRL[RUN] then CTRL[16] is set; otherwise if thread B writes CTRL[RUN] then CTRL[17] is set. These bits cannot be set directly by writing bits 16 or 17 with a **mtctrl** instruction. They are only set when a thread writes CTRL[RUN].
18:30	Reserved	Bits are not implemented; all bits read back zero.
31	RUN	Run state bit.

Additional Information:

- *Book III: PowerPC Operating Environment Architecture, Version 2.02*

PPE Processor Version Register (PVR)

Register Short Name	PVR	**Unit**	XU
Register Type	SPR Read only	**Width**[1]	32 bits
Decimal SPR Number	287	**Access Type**	Read: Privileged
Value at Initial POR	x'0070_0100' (DD1) x'0070_0400' (DD2) x'0070_0500' (DD3.0) x'0070_0501' (DD3.1)	**Set By**	This is a constant value
Additional Information	PowerPC architected register with implementation-specific bits	**Reg. Duplicated for MT**	No

1. The lower 16 bits are described in the datasheet for each DD revision.

Additional Information:

- *Book III: PowerPC Operating Environment Architecture, Version 2.02*

Processor Identification Register (PIR)

Register Short Name	PIR	**Unit**	XU
Register Type	SPR Read only	**Width**	32 bits
Decimal SPR Number	1023	**Access Type**	Read: Privileged
Value at Initial POR	[0:22] Zero [23:30] Set by configuration chain [31] 0 for thread 0 1 for thread 1	**Set By**	Scan initialization during POR Configuration ring
Additional Information	PowerPC architected register with implementation-specific bits	**Reg. Duplicated for MT**	Yes

Additional Information:

- *Book III: PowerPC Operating Environment Architecture, Version 2.02*

CBEA-Compliant Processor Version Register (BP_VR)

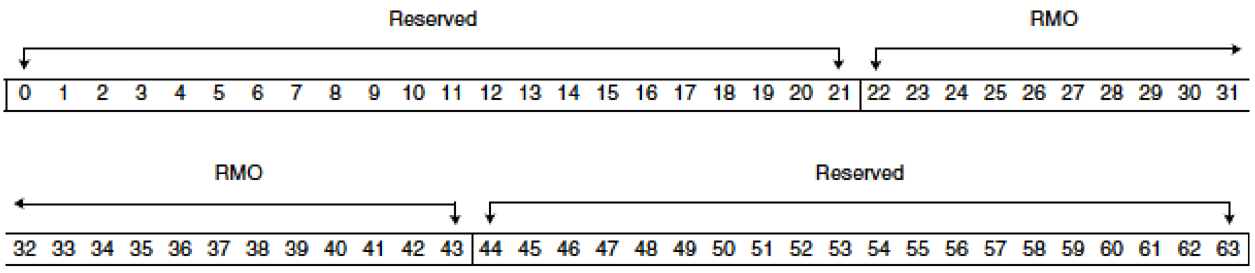
Register Short Name	BP_VR	**Unit**	XU
Register Type	SPR Read only	**Width**	32 bits
Decimal SPR Number	1022	**Access Type**	Read: Privileged
Value at Initial POR	x‘0000_0000’ (DD 1.0) x‘0000_0001’ (DD 1.1) x‘0000_0100’ (DD 2.0) x‘0000_0200’ (DD 3.0) x‘0000_0201’ (DD 3.1)	**Set By**	Hardwired
Additional Information	CBEA architected register	**Reg. Duplicated for MT**	No

Additional Information:

- *Cell Broadband Engine™ Architecture*

Real Mode Offset Register (RMOR)

Register Short Name	RMOR	**Unit**	MMU
Register Type	SPR Read/Write	**Width**	64 bits
Decimal SPR Number	312	**Access Type**	Hypervisor
Value at Initial POR	All bits set to zero	**Set By**	Scan initialization during POR
Additional Information	PowerPC architected register	**Reg. Duplicated for MT**	No



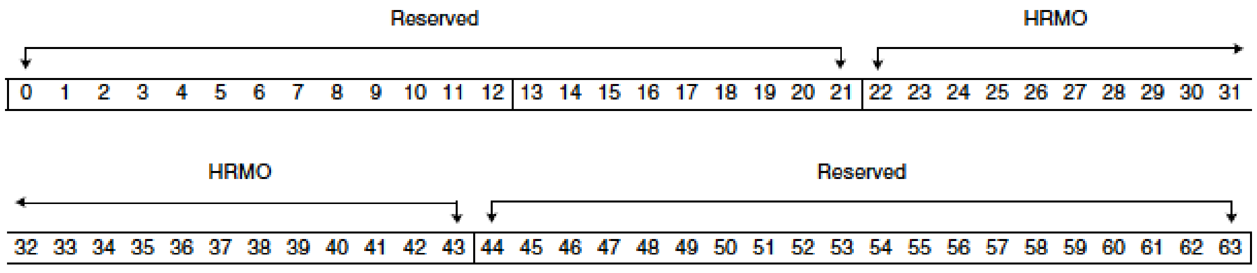
Bit(s)	Field Name	Description
0:21	Reserved	Bits are not implemented; all bits read back zero.
22:43	RMO	Real mode offset The offset from x'0' at which real-mode memory begins.
44:63	Reserved	Bits are not implemented; all bits read back zero.

Additional Information:

- *Book III: PowerPC Operating Environment Architecture, Version 2.02*

Hypervisor Real Mode Offset Register (HRMOR)

Register Short Name	HRMOR	**Unit**	MMU
Register Type	SPR Read/Write	**Width**	64 bits
Decimal SPR Number	313	**Access Type**	Hypervisor
Value at Initial POR	All bits set to zero	**Set By**	Scan initialization during POR
Additional Information	PowerPC architected register	**Reg. Duplicated for MT**	No



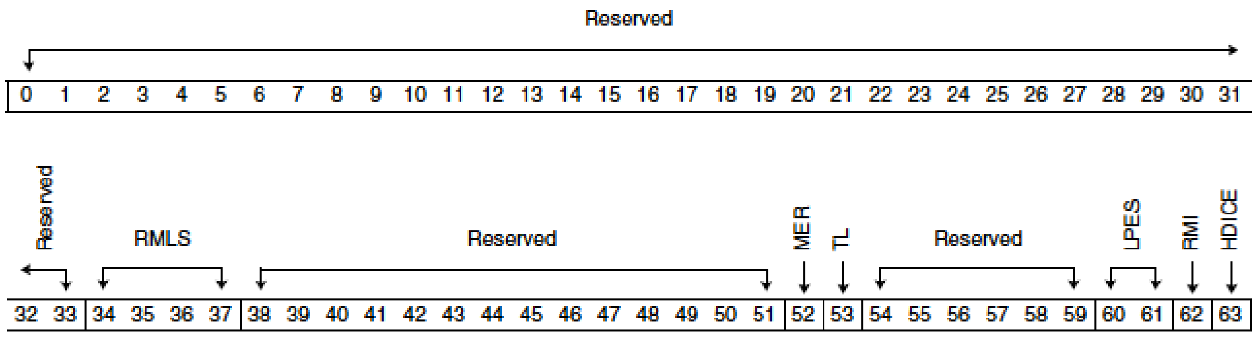
Bit(s)	Field Name	Description
0:21	Reserved	Bits are not implemented; all bits read back zero.
22:43	HRMO	Hypervisor real mode offset The offset from x'0' at which hypervisor real-mode memory begins.
44:63	Reserved	Bits are not implemented; all bits read back zero.

Additional Information:

- *Book III: PowerPC Operating Environment Architecture, Version 2.02*

Logical Partition Control Register (LPCR)

Register Short Name	LPCR	**Unit**	MMU
Register Type	SPR Read/Write	**Width**	64 bits
Decimal SPR Number	318	**Access Type**	Hypervisor
Value at Initial POR	All bits set to zero	**Set By**	Scan initialization during POR
Additional Information	PowerPC architected register	**Reg. Duplicated for MT**	Partially *(See notes in the bit definitions below.)*



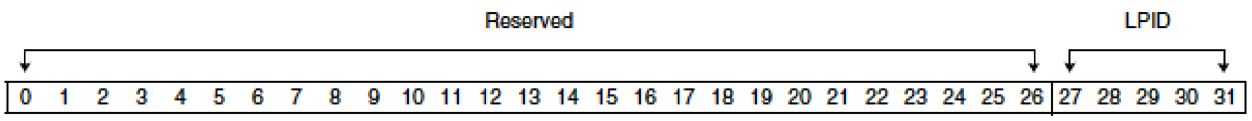
Bit(s)	Field Name	Description
0:33	Reserved	Bits are not implemented; all bits read back zero.
34:37	RMLS	Real mode limit selector. This field is shared by both threads. For additional information, see *Section 8.5 Real Addressing Mode* on page 78.
38:51	Reserved	Bits are not implemented; all bits read back zero.
52	MER	Mediated external exception request (interrupt enable). This field is duplicated per thread. For additional information, see *Section 9.3.3 Mediated External Interrupt* on page 85.
53	TL	TLB load. This field is shared by both threads. 0 TLB loaded by processor 1 TLB loaded by software For additional information, see *Section 8.4.3 Tablewalk* on page 75.
54:59	Reserved	Bits are not implemented; all bits read back zero.
60:61	LPES	Logical partitioning (environment selector). This field is shared by both threads. For more information, see *Section 8.5 Real Addressing Mode* on page 78.
62	RMI	Real-mode caching (cache inhibited). This field is duplicated per thread. For additional information, see *Section 8.5.1 PPE Real-Mode Storage Control Facility* on page 79.
63	HDICE	Hypervisor decrementer interrupt control enable. This field is duplicated per thread. For additional information, see *Section 9 Exceptions and Interrupts*.

Additional Information:

- *Book III: PowerPC Operating Environment Architecture, Version 2.02*

Logical Partition Identity Register (LPIDR)

Register Short Name	LPIDR	**Unit**	MMU
Register Type	SPR Read/Write	**Width**	32 bits
Decimal SPR Number	319	**Access Type**	Hypervisor
Value at Initial POR	All bits set to zero	**Set By**	Scan initialization during POR
Additional Information	PowerPC architected register	**Reg. Duplicated for MT**	No



Bit(s)	Field Name	Description
0:26	Reserved	Bits are not implemented; all bits read back zero.
27:31	LPID	Logical Partition ID

Additional Information:

- *Book III: PowerPC Operating Environment Architecture, Version 2.02*

Programming Note:

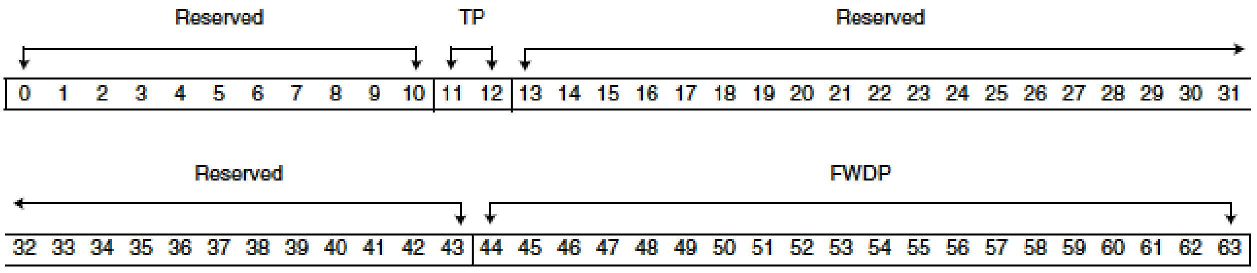
TLB entries are tagged with the LPID when they are created. Therefore, the TLB does not need to be invalidated on a partition context switch.

Thread Status Register Local (TSRL)

This register allows a thread to read its own status.

Register Short Name	TSRL	**Unit**	IU
Register Type	SPR Read/Write	**Width**	64 bits
Decimal SPR Number	896	**Access Type**	Read/Write: Not Privileged
Value at Initial POR	All bits set to zero	**Set By**	Scan initialization during POR
Additional Information	Implementation-specific register	**Reg. Duplicated for MT**[1]	Yes

1. Each thread has a Thread Status Register (TSR). When a thread reads its own TSR register, this register is called the Thread Status Register Local (TSRL). When a thread reads the TSR for the other thread, this register is called the Thread Status Register Remote (TSRR).



Bit(s)	Field Name	Description
0:10	Reserved	Bits are not implemented; all bits read back zero.
11:12	TP	Thread Priority (read/write) '00' Disabled '01' Low '10' Medium '11' High. If a System Reset interrupt is taken, this field is set to '11'. A thread cannot disable itself by attempting to directly set the TP field to ‘00’ (an attempt to do so is ignored). The thread must be disabled by setting the CTRL register appropriately. When in problem state (MSR[PR] = '1'): •If TSCR[UCP] = '0', then the priority cannot be changed. •If TSCR[UCP] = '1', then the priority can be set to Low and Medium. When in priveledged but non-hypervisor state (MSR[HV,PR] = ‘00’): •If TSCR[UCP, PSCTP] = ‘00’, the priority cannot be changed. •If TSCR[UCP, PSCTP] = ‘10’, the priority can be set to Low and Medium. •If TSCR[PSCTP] = '1', the priority can be set to Low, Medium and High. When in hypervisor state (MSR[HV, PR] = ‘10’), the priority can be set to Low, Medium, and High. **Note**: Attempts to set the TP field illegaly is ignored and the priority does not change.
13:43	Reserved	Bits are not implemented; all bits read back zero.
44:63	FWDP	Forward Progress Timer (read only) This field is loaded from TTR[TTIM] each time the current thread completes a PowerPC Architecture instruction. This resets the timer to the maximum count. If the current thread is not disabled (TSRL[TP] = '00'), this field is decremented by one each time an instruction completes on the opposite thread. If TSCR[FPCF] = '1', and the timer reaches 'x00001', then after the next instructions completes, instructions for the opposite thread are flushed and no dispatch slots are given to the opposite thread until one instruction completes on the current thread. This field stops decrementing at x'00001' (the minimum count). This field is Initialized at POR to x'00000' (the maximum count).

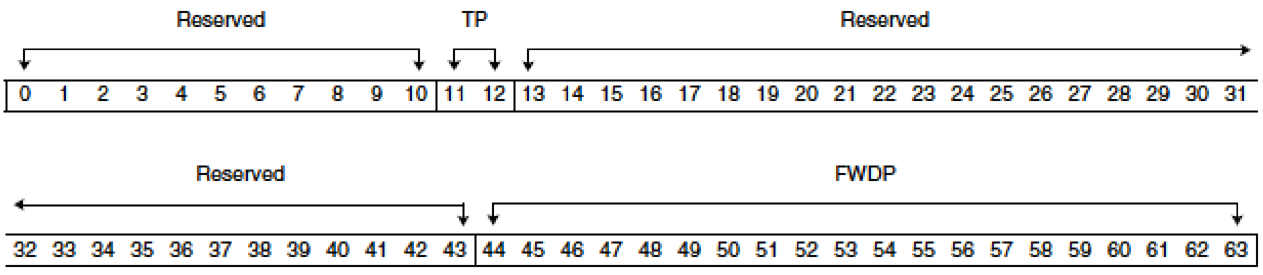
Related Registers: *Thread Status Register Remote (TSRR)* on page 134

Thread Status Register Remote (TSRR)

This register allows a thread to read the status of the other thread.

Register Short Name	TSRR	**Unit**	IU
Register Type	SPR Read Only	**Width**	64 bits
Decimal SPR Number	897	**Access Type**	Not Privileged
Value at Initial POR	All bits set to zero	**Set By**	Scan initialization during POR
Additional Information	Implementation-specific register	**Reg. Duplicated for MT**[1]	Yes

1. Each thread has a Thread Status Register (TSR). When a thread reads its own TSR register, this register is called the Thread Status Register Local (TSRL). When a thread reads the TSR for the other thread, this register is called the Thread Status Register Remote (TSRR).

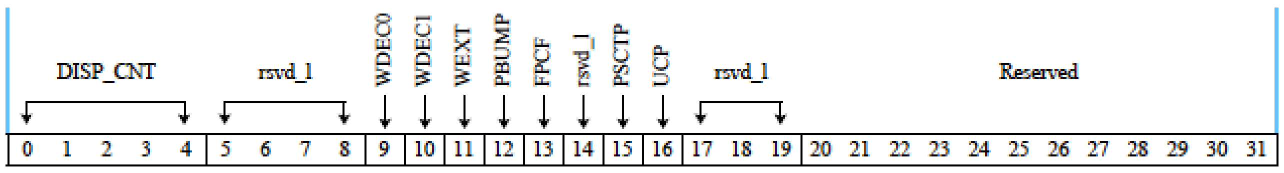


Bit(s)	Field Name	Description
0:10	Reserved	Bits are not implemented; all bits read back zero.
11:12	TP	Thread priority Shows the thread priority of the opposite thread. '00' Disabled '01' Low priority '10' Medium priority '11' High priority
13:43	Reserved	Bits are not implemented; all bits read back zero.
44:63	FWDP	Forward progress timer Shows the Forward Progress Timer counter value of the opposite thread. See the TSRL[FWDP] field description.

Related Registers: *Thread Status Register Local (TSRL)* on page 133

Thread Switch Control Register (TSCR)

Register Short Name	TSCR	**Unit**	IU
Register Type	SPR Read/Write	**Width**	32 bits
Decimal SPR Number	921	**Access Type**	Hypervisor
Value at Initial POR	All bits set to zero	**Set By**	Scan initialization during POR
Additional Information	Implementation-specific registers	**Reg. Duplicated for MT**	No

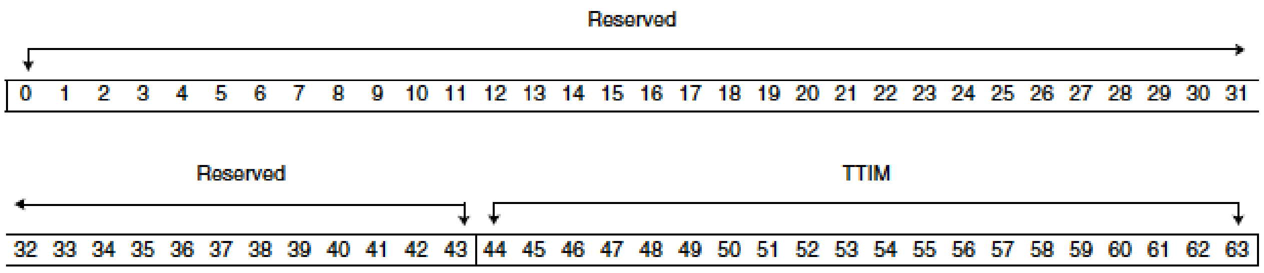


Bit(s)	Field Name	Description
0:4	DISP_CNT	Thread dispatch count Used to control the number of dispatch cycles each thread is given, based on the priority of the thread. A DISP_CNT of '00000' equals 32, which is the maximum dispatch count. During normal operation, this field is recommended to be set to 4.
5:9	rsvd_l	Reserved. Latch bit is implemented; value read is the value written.
9	WDEC0	Decrementer wakeup enable for Thread 0 0 Disabled 1 If the decrementer exception exists, and the corresponding thread is suspended, then the thread is activated.
10	WDEC1	Decrementer wakeup enable for Thread 1 0 Disabled 1 If the decrementer exception exists, and the corresponding thread is suspended, then the thread is activated.
11	WEXT	External Interrupt wakeup enable 0 Disabled 1 If an external interrupt exception exists, and the corresponding thread is suspended, then the thread is activated.
12	PBUMP	Thread priority boost enable 0 Disabled 1 If a system-caused interrupt exception is presented, the corresponding interrupt is not masked, and the priority of the corresponding thread is less than medium, sets the priority of the thread to medium. The hardware internally boosts the priority level to medium when the interrupt is pending. This does not change the value in the TSRL[TP] bits for the affected thread. The internal priority remains boosted to medium until a **mttsrl** or priority changing **nop** instruction occurs. For a discussion of thread priority changes due to interrupts, see *Section 6.6.3.2 Thread Priority* on page 50.
13	FPCF	Forward Progress Count Flush Enable **Note:** This bit only enables or disables the flush from occurring. The forward progress timer does not stop decrementing when set to zero. During normal operation, this bit should be set to '1'.
14	rsvd_l	Reserved. Latch bit is implemented; value read is the value written.
15	PSCTP	Privileged but not hypervisor state change thread priority enable Enables the privileged but not hypervisor state (MSR[HV, PR] = '00') to change priority with "**or** Rn, Rn, Rn" **nop**s or writes to TSRL[TP]. See *Section 6.6.3.2 Thread Priority* on page 50. 0 The privileged state ability to change thread priority is determined by TSCR[UCP]. 1 The privileged state can change thread priority to Low, Medium and High.
16	UCP	Problem State Change Thread Priority Enable Enables the problem state to change priority with "**or** Rn, Rn, Rn" **nop**s or writes to TSRL[TP]. See *Section 6.6.3.2 Thread Priority* on page 50. 0 The problem state can not change thread priority. 1 The problem state can change thread priority to low and medium only.
17:19	rsvd_l	Reserved. Latch bit is implemented; value read is the value written.
20:31	Reserved	Bits are not implemented; all bits read back zero.

Thread Switch Time-Out Register (TTR)

This register is used to ensure forward progress of the instruction dispatch.

Register Short Name	TTR	**Unit**	IU
Register Type	SPR Read/Write	**Width**	64 bits
Decimal SPR Number	922	**Access Type**	Hypervisor
Value at Initial POR	All bits set to zero	**Set By**	Scan initialization during POR
Additional Information	Implementation-specific register	**Reg. Duplicated for MT**	No



Bit(s)	Field Name	Description
0:43	Reserved	Bits are not implemented; all bits read back zero.
44:63	TTIM	Thread time-out flush value A value of x'00000' generates the maximum count. See the description of TSRL[FWDP]. The recommended value for this field is x'04000'. This setting along with the TSCR[FPCF] set to '1' are necessary to guaranty that a forward progress timeout does not occur because one thread prevents the other from completing instructions. If one thread executes 16 K instructions without the other completing one, than the first thread is blocked even if it has higher priority, and the second thread gets full resources to execute an instruction.

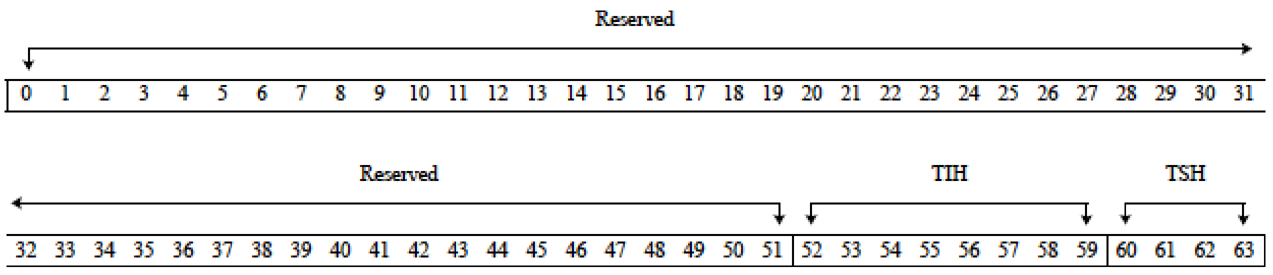
Translation Lookaside Buffer Special Purpose Registers

This section describes the Translation Lookaside Buffer (TLB) Special Purpose Registers (SPRs).

PPE Translation Lookaside Buffer Index Hint Register (PPE_TLB_Index_Hint)

Hardware updates this register when a TLB miss occurs with LPCR[TL] = '1' (software tablewalk mode).

Register Short Name	PPE_TLB_Index_Hint	**Unit**	MMU
Register Type	SPR Read only	**Width**	64 bits
Decimal SPR Number	946	**Access Type**	Read: Privileged
Value at Initial POR	All bits set to zero	**Set By**	Scan initialization during POR
Additional Information	Implementation-specific register	**Reg. Duplicated for MT**	Yes



Bit(s)	Field Name	Description
0:51	Reserved	Bits are not implemented; all bits read back zero.
52:59	TIH	PPE TLB index hint The Index (congruence class) of the TLB that the hardware LRU mechanism would have chosen to replace if hardware TLB updates were enabled (LPCR[TL] = '0').
60:63	TSH	TLB set hint The recommended set for replacement (software replaces entries in correct order to maintain the LRU). Valid values are: '1000' Set 0 '0100' Set 1 '0010' Set 2 '0001' Set 3

Programming Note:

- The PPE_TLB_Index_Hint register is separate from the PPE_TLB_Index to avoid the possibility of hardware changing the index due to a fault while software is updating a TLB entry.
- This implementation supports a 256x4 TLB array. Hence 8 fully-encoded bits [52:59] are used to choose among the 256 rows (or congruence classes) of the TLB and 4 fully-decoded bits [60:63] are used to choose among the 4 columns (or sets).
- The effective address that caused the TLB miss that lead to this register being set can be determined from the DAR.

Related Registers: *PPE Translation-Lookaside Buffer Index Register* (PPE_TLB_Index) on page 138

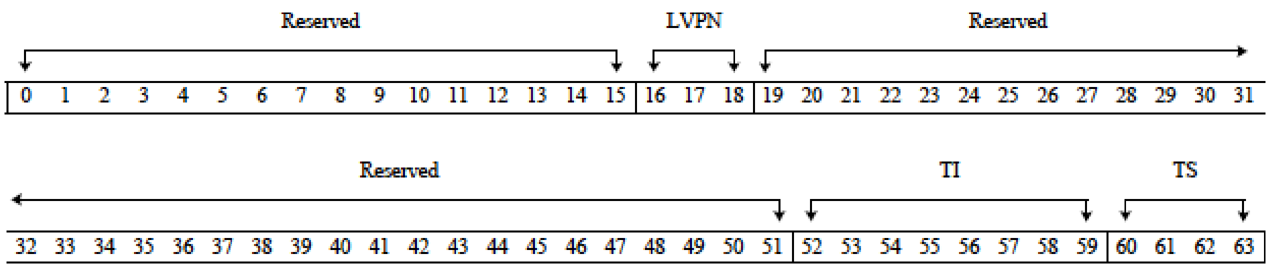
Additional Information:

- *Cell Broadband Engine™ Architecture*

PPE Translation-Lookaside Buffer Index Register (PPE_TLB_Index)

Register Short Name[1]	PPE_TLB_Index	Unit	MMU
Register Type	SPR Read/Write	Width	64 bits
Decimal SPR Number	947	Access Type	Hypervisor
Value at Initial POR	All bits set to zero	Set By	Scan initialization during POR
Additional Information	Implementation-specific register	Reg. Duplicated for MT	No

1. This register is read for diagnostic purposes.



Bit(s)	Field Name	Description
0:15	Reserved	Bits are not implemented; all bits read back zero.
16:18	LVPN	Lower Virtual Page Number LVPN[0:2] which corresponds to VPN[57:59]. **Note:** AVPN[0:56] = VPN[0:56] VPN = AVPN || LVPN. The PPU only implements LVPN[0:2], since LVPN[3:12-p] is implied by the INDEX field of the PPE_TLB_INDEX register.
19:51	Reserved	Bits are not implemented; all bits read back zero.
52:59	TI	PPE TLB Index The fully-encoded index of the TLB chosen for replacement.
60:63	TS	TLB set The set chosen for replacement. The following are valid set combinations: 1000 – set 0 0100 – set 1 0010 – set 2 0001 – set 3 Setting multiple bits causes multiple sets in the TLB to written with identical data. This is not recommended as it makes inefficient use of the TLB.

Programming Note:

- This implementation supports a 256 x 4 TLB array. Hence 8 fully-encoded bits [52:59] are used to choose among the 256 rows (or congruence classes) of the TLB and 4 fully-decoded bits [60:63] are used to choose among the 4 columns (or sets).
- This register is read by the MMU hardware when an **mtspr** or **mfspr** instruction is executed with a target address of either the TLB_VPN or TLB_RPN. This allows the MMU to know which entry of the TLB software wishes to update. Software writes to this register to indicate the desired entry to replace. The register is also readable for debug purposes. Subsequent writes to the PPE_TLB_VPN and PPE_TLB_RPN are based on the last written value to this register.

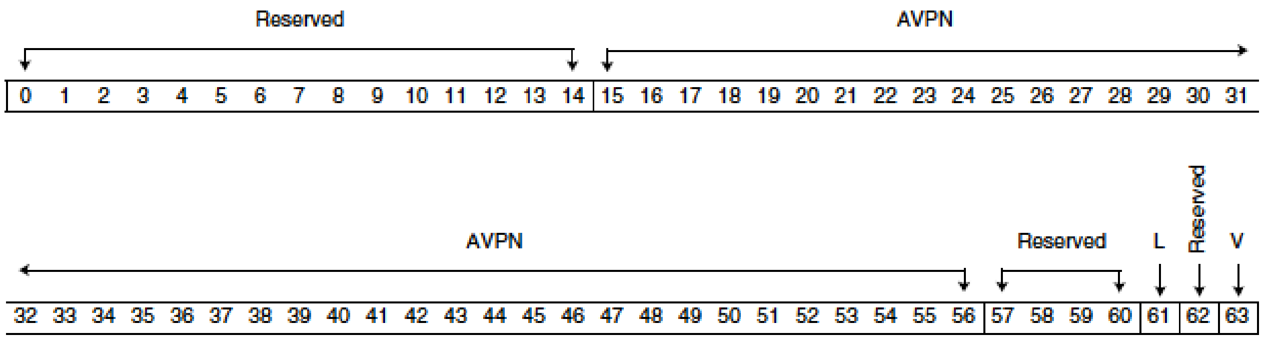
Related Registers: *PPE Translation Lookaside Buffer Index Hint Register* (PPE_TLB_Index_Hint) on page 137

Additional Information:

- *Cell Broadband Engine™ Architecture*

PPE Translation-Lookaside Buffer Virtual-Page Number Register (PPE_TLB_VPN)

Register Short Name	PPE_TLB_VPN	**Unit**	MMU
Register Type	SPR Read/Write	**Width**	64 bits
Decimal SPR Number	948	**Access Type**	Hypervisor
Value at Initial POR	All bits set to zero	**Set By**	Scan initialization during POR
Additional Information	Implementation-specific register	**Reg. Duplicated for MT**	No



Bit(s)	Field Name	Description
0:14	Reserved	Bits are not implemented; all bits read back zero.
15:56	AVPN	Abbreviated virtual page number (AVPN) AVPN = VPN[15:56] For a description of AVPN, see *Book III: PowerPC Operating Environment Architecture, Version 2.02* **Note:** When reading a 16 MB TLB entry, bit 56 of the AVPN is undefined and software should ignore it.
57:60	Reserved	Bits are not implemented; all bits read back zero.
61	L	Large-page mode 0 4KB page 1 Large page
62	Reserved	Bits are not implemented; all bits read back zero.
63	V	Valid bit 0 Invalid 1 Valid

Programming Note:

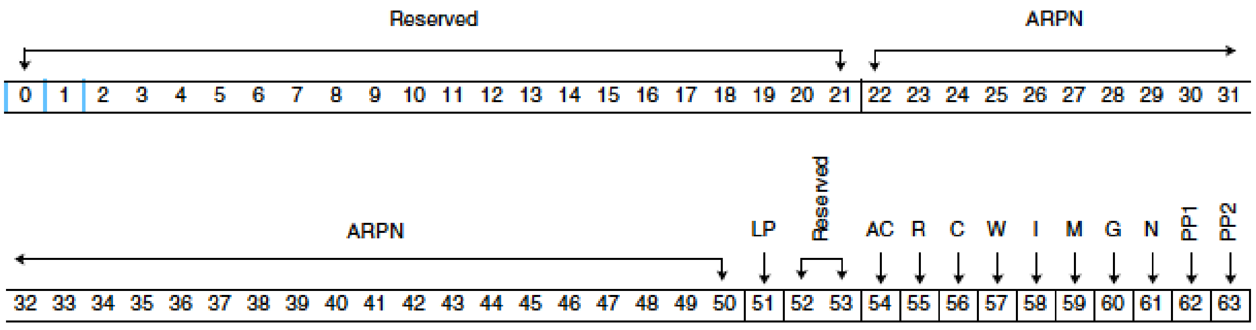
- If the VPN is being invalidated to change the protection attributes of a page, or to *steal* the page, a TLB Invalidate Entry command must be issued to invalidate any cache of the effective-to-real address translation that may be associated with the TLB entry being invalidated.
- This register acts as a place-holder for the TLB entry pointed to by PPE_TLB_Index. Changing the value of PPE_TLB_Index implicitly changes the value of this register to match the contents of the corresponding TLB array entry pointed to by PPE_TLB_Index.
- This register is meant to be written as part of a sequence of instructions. For details, see *Section 12.3 TLB Management Instructions and ERAT Coherency* on page 125.

Additional Information:

- *Cell Broadband Engine™ Architecture*

PPE Translation-Lookaside Buffer Real-Page Number Register (PPE_TLB_RPN)

Register Short Name	PPE_TLB_RPN	**Unit**	MMU
Register Type	SPR Read/Write	**Width**	64 bits
Decimal SPR Number	949	**Access Type**	Hypervisor
Value at Initial POR	All bits set to zero	**Set By**	Scan initialization during POR
Additional Information	Implementation-specific register	**Reg. Duplicated for MT**	No



Bit(s)	Field Name	Description
0:21	Reserved	Bits are not implemented; all bits read back zero.
22:50	ARPN	Abbreviated real page number ARPN = RPN[22:50] To obtain the full 30-bit RPN, the ARPN is combined with the LP field.
51	LP	Large page size selector If PPE_TLB_VPN[L] = '1', then bit 51 is used as page size selector (see HID6[LB] in *Hardware Implementation Register 6 (HID6)* on page *154*), otherwise bit [51] = RPN[51].
52:53	Reserved	Bits are not implemented; all bits read back zero.
54	AC	Address Compare
55	R	Reference This bit is treated as always '1' by the implementation. Any attempt to set it to '0' is ignored.
56	C	Change
57	W	Write-through This bit is forced to '0' in this implementation.
58	I	Caching Inhibited
59	M	Memory coherency bit Memory is always coherent on this processor so this value is forced to '1'. **Note:** Reference and Change bit updates are done with M = '1'. Software should not set the Page Table Entry (PTE) M bit to '0' since it may be implicitly overwritten by a reference or change bit update.
60	G	Guarded
61	N	No execute 0 Execute page 1 No-execute page
62	PP1	Page protection bit 1 for tags inactive mode
63	PP2	Page protection bit 2 for tags inactive mode

Programming Note:

- This implementation supports two concurrent large page sizes. The selection between the two large page sizes is controlled by the LP field when the L bit is set in the PPE_TLB_VPN. The address of the real page (that is, the Real Page Number or RPN) is formed by concatenating the ARPN with a zero when the L bit is set. Since the RPN must be on a page size boundary, software must set some low-order bits to zero when L is set or the results are undefined. For 64 KB, 1 MB, and 16 MB pages, the low-order 4, 8, and 12 bits of the RPN respectively are zero. If the L bit is not set in the PPE_TLB_VPN, the RPN is formed by concatenating the ARPN with the LP field, and the page size is 4KB.

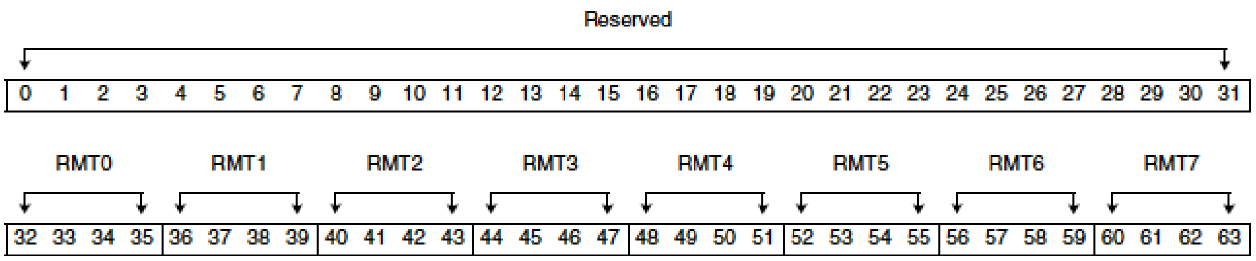
- This register acts as a place-holder for the TLB entry pointed to by PPE_TLB_Index. Changing the value of PPE_TLB_Index implicitly changes the value of this register to match the contents of the corresponding TLB array entry pointed to by PPE_TLB_Index.
- This register is meant to be written as part of a sequence of instructions. For details, see *Section 12.3 TLB Management Instructions and ERAT Coherency* on page 125.

Additional Information:

- *Cell Broadband Engine™ Architecture*

PPE Translation-Lookaside Buffer RMT Register (PPE_TLB_RMT)

Register Short Name	PPE_TLB_RMT	**Unit**	MMU
Register Type	SPR Read/Write	**Width**	64 bits
Decimal SPR Number	951	**Access Type**	Hypervisor
Value at Initial POR	All bits set to zero	**Set By**	Scan initialization during POR
Additional Information	Implementation-specific register	**Reg. Duplicated for MT**	No



Bit(s)	Field Name	Description
0:31	Reserved	Bits are not implemented; all bits read back zero.
32:35	RMT0	Entry 0 of the replacement management table (RMT)
36:39	RMT1	Entry 1 of the RMT
40:43	RMT2	Entry 2 of the RMT
44:47	RMT3	Entry 3 of the RMT
48:51	RMT4	Entry 4 of the RMT
52:55	RMT5	Entry 5 of the RMT
56:59	RMT6	Entry 6 of the RMT
60:63	RMT7	Entry 7 of the RMT

Programming Note:

- Each RMT entry is 4 bits which are fully-decoded and correspond to a set in the TLB. If an effective address matches a range-register, then the TLB considers the corresponding RMT entry for this range when replacing entries in the TLB. Each bit of the RMT entry can be thought of as a set-enabler, indicating that, when set to ‘1’, the corresponding set of the TLB is a valid entry to replace if the translation requires a TLB update to occur (e.g. when the translation page table entry does not currently reside in the TLB). If multiple sets are indicated, they are replaced in priority order beginning with invalid entries first, and then valid entries in a left-to-right fashion.

Additional Information:

- *Cell Broadband Engine™ Architecture*

Data Address Range SPRs

Data Address Range Start Register 0 (DRSR0)

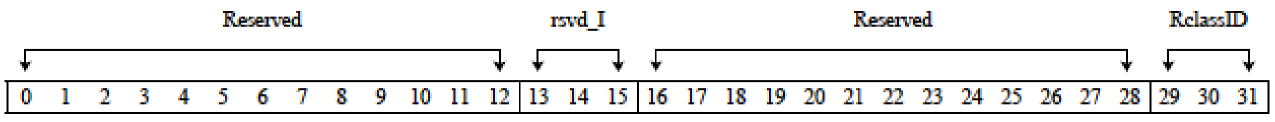
Register Short Name	DRSR0	**Unit**	XU
Register Type	SPR Read/Write	**Width**	64 bits
Decimal SPR Number	952	**Access Type**	Hypervisor
Value at Initial POR	All bits set to zero	**Set By**	Scan initialization during POR
Additional Information	CBEA architected register	**Reg. Duplicated for MT**	Yes Each thread has a DRSR0 and a DRSR1.

Data Range Mask Register 0 (DRMR0)

Register Short Name	DRMR0	**Unit**	XU
Register Type	SPR Read/Write	**Width**	64 bits
Decimal SPR Number	953	**Access Type**	Hypervisor
Value at Initial POR	All bits set to zero	**Set By**	Scan initialization during POR
Additional Information	CBEA architected register	**Reg. Duplicated for MT**	Yes Each thread has a DRMR0 and a DRMR1.

Data Class ID Register 0 (DCIDR0)

Register Short Name	DCIDR0	**Unit**	XU
Register Type	SPR Read/Write	**Width**	32 bits
Decimal SPR Number	954	**Access Type**	Hypervisor
Value at Initial POR	All bits set to zero	**Set By**	Scan initialization during POR
Additional Information	Implementation-specific register	**Reg. Duplicated for MT**	Yes Each thread has a DCIDR0 and a DCIDR1.



The following are the implementation-specific bits for this register:

Bit(s)	Field Name	Description
0:12	Reserved	Bits are not implemented; all bits read back zero.
13:15	rsvd_I	Reserved. Latch bit is implemented; value read is the value written.
16:28	Reserved	Bits are not implemented; all bits read back zero.
29:31	RclassID	RclassID

Data Range Start Register 1 (DRSR1)

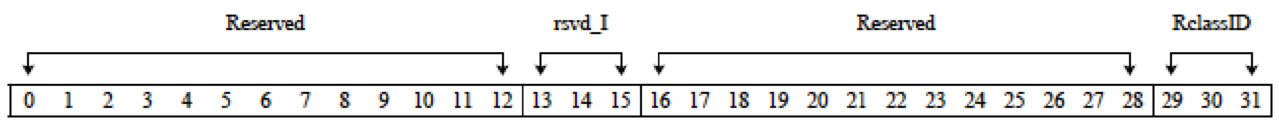
Register Short Name	DRSR1	**Unit**	XU
Register Type	SPR Read/Write	**Width**	64 bits
Decimal SPR Number	955	**Access Type**	Hypervisor
Value at Initial POR	All bits set to zero	**Set By**	Scan initialization during POR
Additional Information	CBEA architected register	**Reg. Duplicated for MT**	Yes Each thread has a DRSR0 and a DRSR1.

Data Range Mask Register 1 (DRMR1)

Register Short Name	DRMR1	**Unit**	XU
Register Type	SPR Read/Write	**Width**	64 bits
Decimal SPR Number	956	**Access Type**	Hypervisor
Value at Initial POR	All bits set to zero	**Set By**	Scan initialization during POR
Additional Information	CBEA architected register	**Reg. Duplicated for MT**	Yes Each thread has a DRMR0 and a DRMR1.

Data Class ID Register 1 (DCIDR1)

Register Short Name	DCIDR1	**Unit**	XU
Register Type	SPR Read/Write	**Width**	32 bits
Decimal SPR Number	957	**Access Type**	Hypervisor
Value at Initial POR	All bits set to zero	**Set By**	Scan initialization during POR
Additional Information	Implementation-specific register	**Reg. Duplicated for MT**	Yes Each thread has a DCIDR0 and a DCIDR1.



Bit(s)	Field Name	Description
0:12	Reserved	Bits are not implemented; all bits read back zero.
13:15	rsvd_I	Reserved. Latch bit is implemented; value read is the value written.
16:28	Reserved	Bits are not implemented; all bits read back zero.
29:31	RclassID	RclassID

Instruction Range SPRs

Instruction Range Start Register 0 (IRSR0)

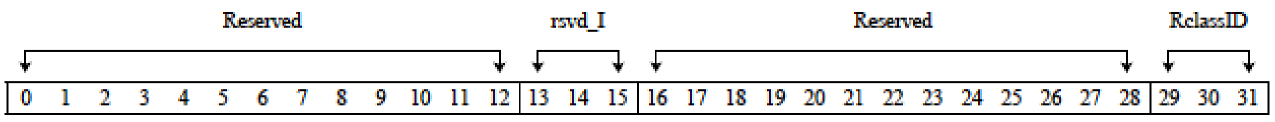
Register Short Name	IRSR0	**Unit**	IU
Register Type	SPR Read/Write	**Width**	64 bits
Decimal SPR Number	976	**Access Type**	Hypervisor
Value at Initial POR	All bits set to zero	**Set By**	Scan initialization during POR
Additional Information	CBEA architected register	**Reg. Duplicated for MT**	Yes Each thread has a IRSR0 and a IRSR1.

Instruction Range Mask Register 0 (IRMR0)

Register Short Name	IRMR0	**Unit**	IU
Register Type	SPR Read/Write	**Width**	64 bits
Decimal SPR Number	977	**Access Type**	Hypervisor
Value at Initial POR	All bits set to zero	**Set By**	Scan initialization during POR
Additional Information	CBEA architected register	**Reg. Duplicated for MT**	Yes Each thread has a IRMR0 and a IRMR1.

Instruction Class ID Register 0 (ICIDR0)

Register Short Name	ICIDR0	**Unit**	IU
Register Type	SPR Read/Write	**Width**	32 bits
Decimal SPR Number	978	**Access Type**	Hypervisor
Value at Initial POR	All bits set to zero	**Set By**	Scan initialization during POR
Additional Information	Implementation-specific register	**Reg. Duplicated for MT**	Yes Each thread has a ICIDR0 and a ICIDR1.



Bit(s)	Field Name	Description
0:12	Reserved	Bits are not implemented; all bits read back zero.
13:15	rsvd_I	Reserved. Latch bit is implemented; value read is the value written.
16:28	Reserved	Bits are not implemented; all bits read back zero.
29:31	RclassID	RclassID

Instruction Range Start Register 1 (IRSR1)

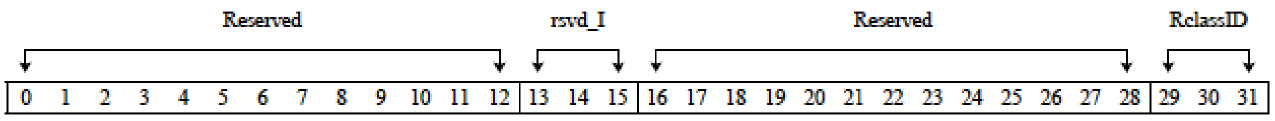
Register Short Name	IRSR1	**Unit**	IU
Register Type	SPR Read/Write	**Width**	64 bits
Decimal SPR Number	979	**Access Type**	Hypervisor
Value at Initial POR	All bits set to zero	**Set By**	Scan initialization during POR
Additional Information	CBEA architected register	**Reg. Duplicated for MT**	Yes Each thread has a IRSR0 and a IRSR1.

Instruction Range Mask Register 1 (IRMR1)

Register Short Name	IRMR1	**Unit**	IU
Register Type	SPR Read/Write	**Width**	64 bits
Decimal SPR Number	980	**Access Type**	Hypervisor
Value at Initial POR	All bits set to zero	**Set By**	Scan initialization during POR
Additional Information	CBEA architected register	**Reg. Duplicated for MT**	Yes Each thread has a IRMR0 and a IRMR1.

Instruction Class ID Register 1 (ICIDR1)

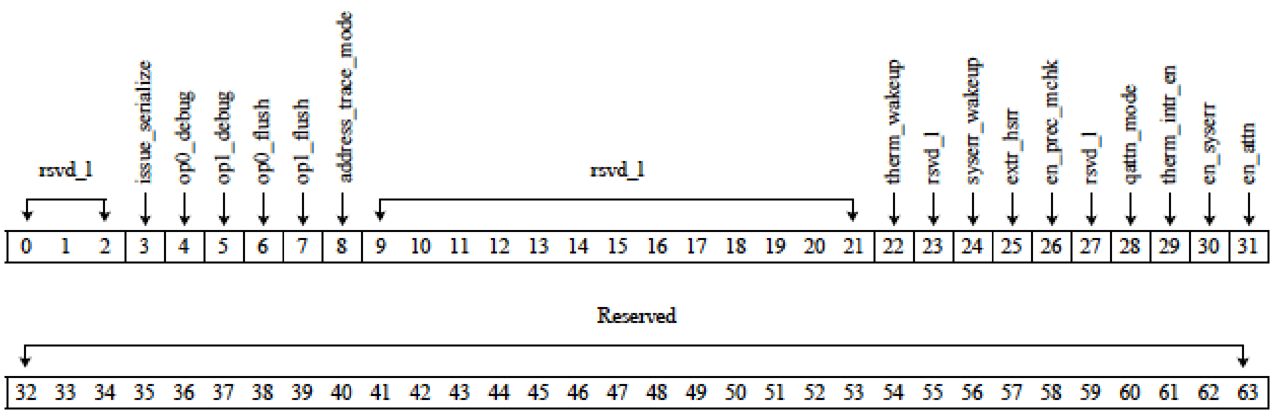
Register Short Name	ICIDR1	**Unit**	IU
Register Type	SPR Read/Write	**Width**	32 bits
Decimal SPR Number	981	**Access Type**	Hypervisor
Value at Initial POR	All bits set to zero	**Set By**	Scan initialization during POR
Additional Information	Implementation-specific register	**Reg. Duplicated for MT**	Yes Each thread has a ICIDR0 and a ICIDR1.



Bit(s)	Field Name	Description
0:12	Reserved	Bits are not implemented; all bits read back zero.
13:15	rsvd_I	Reserved. Latch bit is implemented; value read is the value written.
16:28	Reserved	Bits are not implemented; all bits read back zero.
29:31	RclassID	RclassID

Hardware Implementation Register 0 (HID0)

Register Short Name	HID0	**Unit**	IU
Register Type	SPR Read/Write	**Width**	64 bits
Decimal SPR Number	1008	**Access Type**	Hypervisor
Value at Initial POR	All bits set to zero	**Set By**	Scan initialization during POR
Additional Information	Implementation-specific register	**Reg. Duplicated for MT**	No



Bit(s)	Field Name	Description
0:2	rsvd_l	Reserved. Latch bit is implemented; value read is the value written.
3	issue_serialize	Issue Serialize Mode 0 Normal operation 1 Next instruction is not issued until all previous instructions have completed. (no dual-issue either)
4	op0_debug	Opcode compare #0 takes maintenance interrupt on opcode compare match. This is a hardware debug facility that is not visible to software. The IU supports two opcode compare facilities that are 32-bit compare under mask control. A hit is detected when all bits are to be compared as determined by the bit-wise mask bits are equal. Compare is not thread based. Action taken at commit point of matching instruction. 0 Does not take maintenance interrupt 1 Opcode compare #0 takes a maintenance interrupt on an opcode compare match
5	op1_debug	Opcode compare #1 takes maintenance interrupt on opcode compare match. This is a hardware debug facility that is not visible to software. 0 Does not take maintenance interrupt 1 Opcode compare #1 takes a maintenance interrupt on an opcode compare match
6	op0_flush	Opcode compare #0 causes an internal flush to that thread 0 Does not cause internal flush 1 Opcode compare #0 causes an internal flush to that thread
7	op1_flush	Opcode compare #1 causes an internal flush to that thread 0 Does not cause internal flush 1 Opcode compare #1 causes an internal flush to that thread
8	address_trace_mode	Address Trace Mode. 0 Every time address trace event occurs, the address for the new event is sent to the trace array. The partial address of older event is recorded. 1 Whole 64-bit address is sent to the trace array every time. Events occurring while writing a 64-bit address are ignored.
9:21	rsvd_l	Reserved. Latch bit is implemented; value read is the value written.
22	therm_wakeup	Enable thermal management interrupt to wakeup suspended thread **Note:** Wakeup occurs even if HID0[therm_intr_en] = '0'.
23	rsvd_l	Reserved. Latch bit is implemented; value read is the value written.

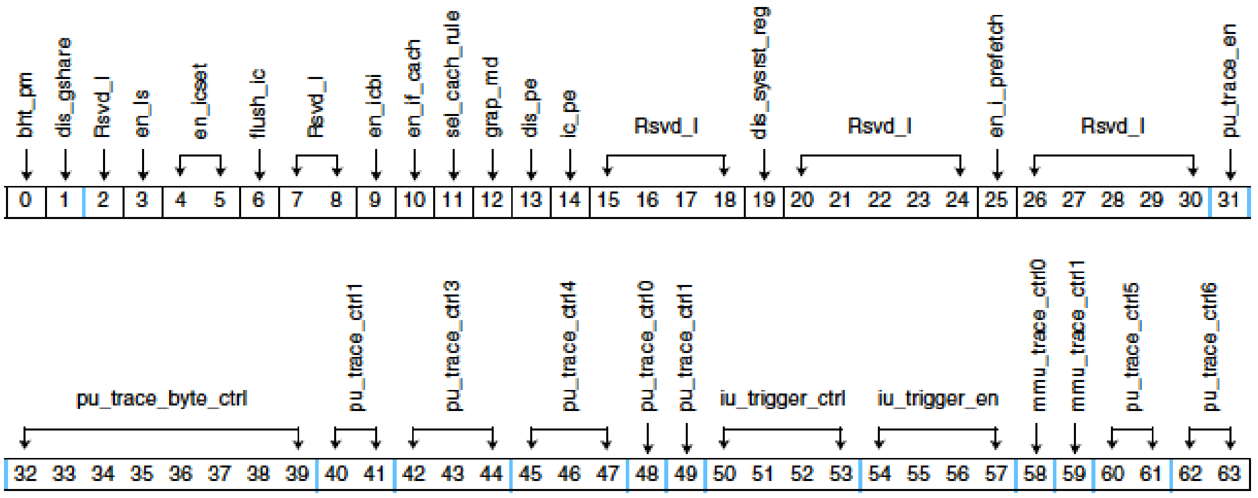
Bit(s)	Field Name	Description
24	syserr_wakeup	Enable system error interrupt to wakeup suspended thread Allows the system error interrupt to wake up either thread if it is dormant. When a system error interrupt is received, if this bit is enabled the interrupt wakes up the thread that the interrupt was intended for (could be both threads). For example. if thread 0 is dormant, thread 1 is active, syserr_wakeup is set, and the interrupt is for thread 0, then thread 0 is awakened (the active thread (thread1) is unaffected since it is already awake). If both threads are dormant and if syserr_wakeup is set, the interrupt awakens both threads. **Note:** Wakeup occurs even if HID0[syserr_wakeup] = '0'.
25	extr_hsrr	Enable extended external interrupt The PowerPC architecture specifies that external interrupts use the HSRR0 and HSRR1 for save and restore of external interrupts when LPCR[LPES0] = '0'. This bit also enables the Mediated External Interrupt feature. For additional information, see *Section 9.3.3 Mediated External Interrupt* on page 85. 0 Normal operation (direct external interrupts) 1 Enabled (mediated external interrupt enabled) **Note:** In the case of mediated external interrupts, setting the LPCR[MER] before disabling a thread with an **mtctrl** instruction awakens the thread if the TSCR[WEXT] is set, even if HID0[25]='0', which normally disables mediated external interrupts.
26	en_prec_mchk	Enable Precise Machine Check **Note:** All loads to caching-inhibited (I='1') space cause the thread to be blocked at dispatch until data is returned for the load.
27	rsvd_l	Reserved. Latch bit is implemented; value read is the value written.
28	qattn_mode	Service processor control 0 ATTN only inactivates the thread issuing the instruction. 1 ATTN on one instruction also inactivates the other thread and causes it to go into maintenance.
29	therm_intr_en	Master thermal management interrupt enable Clearing this bit disables all thermal management interrupts regardless of MSR state. 0 Disables all thermal management interrupts regardless of MSR state 1 Enabled
30	en_syserr	Enable system errors System errors generated from outside the PPE. 0 Disabled 1 Enabled
31	en_attn	Enable Attention instruction (enable support processor **attn** instruction) This is a hardware debug facility that is not visible to software. This is used to enable the **attn** instruction to quiesce the processor. If this bit is disabled, the **attn** instruction is treated as an illegal instruction.
32:63	Reserved	Bits are not implemented; all bits read back zero.

Programming Note:

- After POR, this register is set to x’0000_0000_0000_0000’. In normal operation, the hypervisor should set this register to its preferred value of x’0000_0047_0000_0000’ before booting the system.
- No changes after POR to this register are necessary in typical operating mode. The only reason to change any field of this register is for diagnostic purposes in a lab environment.

Hardware Implementation Register 1 (HID1)

Register Short Name	HID1	**Unit**	IU
Register Type	SPR Read/Write	**Width**	64 bits
Decimal SPR Number	1009	**Access Type**	Hypervisor
Value at Initial POR	All bits set to zero	**Set By**	Scan initialization during POR
Additional Information	Implementation-specific register	**Reg. Duplicated for MT**	No



Bit(s)	Field Name	Description
0	bht_pm	Branch History Table Prediction Mode 0 Static prediction. A conditional branch instruction is always predicted as not taken. 1 BHT is used for branch prediction.
1	dis_gshare	Disable GShare Branch Prediction 0 The global history table is active 1 Forces global history bits to always zero (which disables the global history table).
2	rsvd_I	Reserved. Latch bit is implemented; value read is the value written.
3	en_ls	Enable Link Stack 0 Disable link stack. The predicted branch target address is always x'0'. 1 Enable link stack. All four entries of Link Stack are used.
4:5	en_icway	Enable Instruction Cache way (one bit per way) '00' Cache disabled '10' Way A enabled '01' Way B enabled '11' Cache enabled **Note:** The instruction L1 cache is thread independent, that is, all instructions access the same cache. The A and B refer to the *way* or *set* (a product of two-way associativity in the cache).
6	flush_ic	Flush Instruction Cache This bit can be used to flush ICache. When this bit is changed from '0' to '1', hardware detects this change and flushed the entire ICache. **Note**: • Software has to reset this bit after it is set (this is called a *sticky* bit). This bit has to be set to '0' before setting it to '1' to trigger a flush, which is positive edge triggered. • The change state only occurs in a set that is enabled. If set is disabled, then there is nothing to flush (it is already marked as invalid or already flushed).
7:8	rsvd_I	Reserved. Latch bit is implemented; value read is the value written.

Bit(s)	Field Name	Description
9	en_icbi	Enable "forced icbi match" mode. In this mode, whenever an instruction cache block invalidate (icbi) is presented to the processor, the eight entries in the I-cache that corresponds to the real address (least significant 12 bits) of the line are invalidated. Eight entries are invalidated as there are four congruence classes per way where an entry can be stored since bits 50:51 of the EA (not RA) are used to index the Instruction Cache. 0 Disable 1 Enable
10	en_if_cach	Enable Instruction Fetch Cacheability Control 0 All instruction fetch accesses are treated as caching inhibited (regardless of the state of the page table I-bit). 1 Instruction fetch cacheability is controlled by the state of the page table I-bit.
11	sel_cach_rule	Select which cacheabilty control rule to use. 0 Configuration ring 1 HID register
12	grap_md	Graphics Rounding Mode The VXU unit operates in Graphics Rounding Mode. See *Section 12.1.2 VXU Graphics Rounding Mode* on page 116 for details.
13	dis_pe	Disable parity error reporting and recovery 0 Normal operation 1 Disable
14	ic_pe	Force instruction cache parity error 0 Normal operation 1 Inject parity error into I-Cache **Note:** • Software has to reset this bit after it is set (this is called a *sticky* bit). This bit has to be set to '0' before setting it to '1' to trigger a parity error, which is positive-edge triggered.
15:18	rsvd_l	Reserved. Latch bit is implemented; value read is the value written.
19	dis_ sysrst_reg	Disable config ring system reset interrupt address register 0 Enable (jump to configuration ring register value) 1 Disable (jump to x'100') This only applies to thread 0. For thread 1, always jump to x'100' on a system reset interrupt.
20:24	rsvd_l	Reserved. Latch bit is implemented; value read is the value written.
25	en_i_prefetch	Enable Instruction Prefetch 0 Normal operation 1 Enable
26:30	rsvd_l	Reserved. Latch bit is implemented; value read is the value written.
31	pu_trace_en	Enable PPU Performance Monitor/Debug Bus 0 PPU Bus is Disabled. Debug Bus Latches do not clock functionality. 1 PPU Bus is Enabled. Debug Bus Latches clock functionality.
32:39	pu_trace_byte_ctrl	Byte Enables for PPU Performance Monitor Bus/Global Debug Bus 0 PPU Bus is Disabled for Byte *x* 1 PPU Bus is Enabled for Byte *x*
40:41	pu_trace_ctrl2	PPU Trace Bus [0:63] Output Control '00' Disable Trace Bus [0:63] (place zeros on bus) '01' Select IU [0:63] Trace Bus '10' Select XU [0:63] Trace Bus '11' Select VSU [0:63] Trace Bus
42:44	pu_trace_ctrl3	PPU Trace Bus [64:95] Output Control '000' Disable Trace Bus [64:95] (place zeros on bus) '001' Select XU [0:31] '010' Select XU [64:95] '011' Select VSU [0:31] '100' Select VSU [64:95] '101' Select IU [64:95]
45:47	pu_trace_ctrl4	PPU Trace Bus [96:127] Output Control '000' Disable Trace Bus [96:127] (place zeros on bus) '001' Select VSU [32:63] '010' Select VSU [96:127] '011' Select XU [32:63] '100' Select XU [96:127] '101' Select IU [96:127]

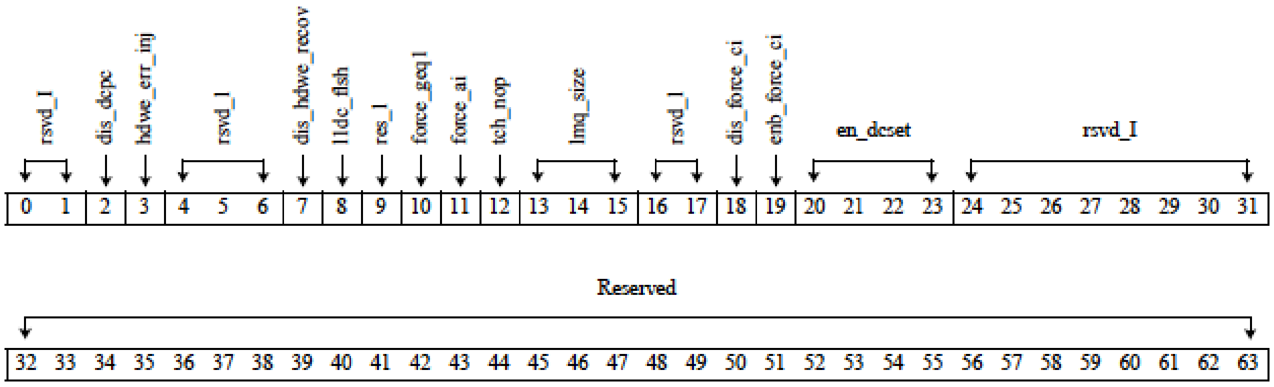
Bit(s)	Field Name	Description
48	pu_trace_ctrl0	PPU Trace Bus [0:63] Output Control 0 Place PPU Trace Bus [0:63] on [0:63] output 1 Place PPU Trace Bus [64:127] on [0:63] output
49	pu_trace_ctrl1	PPU Trace Bus [64:127] Output Control 0 Place PPU Trace Bus [64:127] on [64:127] output 1 Place PPU Trace Bus [0:63] on [64:127] output
50:53	iu_trigger_ctrl	IU Trigger Bus Control 0 IABR Match Thread0 should be placed on IU Trigger Bus bit *x* 1 IABR Match Thread1 should be placed on IU Trigger Bus bit *x* Bit [50] controls Trigger Bus Bit [0] Bit [51] controls Trigger Bus Bit [1] Bit [52] controls Trigger Bus Bit [2] Bit [53] controls Trigger Bus Bit [3]
54:57	iu_trigger_en	IU Trigger Bus Enable 0 Pass XU Trigger But onto PPU Trigger Bus for bit *x* 1 Pass IU Trigger Bus onto PPU Trigger Bus for bit *x* Bit [54] controls Trigger Bus Bit [0] Bit [55] controls Trigger Bus Bit [1] Bit [56] controls Trigger Bus Bit [2] Bit [57] controls Trigger Bus Bit [3]
58	mmu_trace_ctrl0	MMU Ramp controls for PPU Trace Bus [0:31] 0 Do not place MMU Trace Bus on final PPU trace Bus output bits [0:31] 1 Place MMU Trace Bus on final PPU trace Bus output bits [0:31]
59	mmu_trace_ctrl1	MMU Ramp controls for PPU Trace Bus [64:95] 0 Do not place MMU Trace Bus on final PPU trace Bus output bits [64:95] 1 Place MMU Trace Bus on final PPU trace Bus output bits [64:95]
60:61	pu_trace_ctrl5	PPU Trace Bus [0:31] Output Control ‘00’ Place PPU Trace Bus [0:31] on [0:31] output ’01’ Place PPU Trace Bus [32:63] on [0:31] output ‘10’ Place PPU Trace Bus [64:95] on [0:31] output
62:63	pu_trace_ctrl6	PPU Trace Bus [32:63] Output Control ‘00’ Place PPU Trace Bus [32:63] on [32:63] output ’01’ Place PPU Trace Bus [0:31] on [32:63] output ‘10’ Place PPU Trace Bus [96:127] on [32:63] output

Programming Note:

- After POR, this register is set to x’0000_0000_0000_0000’. In normal operation, the hypervisor should set this register to its preferred value of x’9C30_1040_0000_0000’ before booting the system.

Hardware Implementation Register 4 (HID4)

Register Short Name	HID4	**Unit**	XU
Register Type	SPR Read/Write	**Width**	64 bits
Decimal SPR Number	1012	**Access Type**	Hypervisor
Value at Initial POR	All bits set to zero	**Set By**	Scan initialization during POR
Additional Information	Implementation-specific register	**Reg. Duplicated for MT**	No



Bit(s)	Field Name	Description
0:1	rsvd_I	Reserved. Latch bit is implemented; value read is the value written.
2	dis_dcpc	Disable data cache parity checking. Prevents Fault Isolation Register (FIR) updates and recovery. 0 Normal operation. 1 Disable data cache parity checking.
3	hdwe_err_inj	Inject parity error into cache. Forces a 1-bit error somewhere in both 32-byte blocks that are being reloaded to the data cache. Used for bring up. 0 Normal operation. 1 Inject parity error into cache.
4:6	rsvd_I	Reserved. Latch bit is implemented; value read is the value written.
7	dis_hdwe_recov	Disable data cache parity error hardware recovery FIR bits are still reported to the MMU to cause machine check/checkstop 0 Normal operation. 1 Disable data cache parity error hardware recovery.
8	l1dc_flsh	L1 Data Cache flash invalidate 0 Normal operation 1 All sectors set to invalid and held invalid. (Implemented as edge detect) **Note:** Software has to reset this bit after it is set (this is called a *sticky* bit). This bit has to be set to '0' before setting it to '1' to trigger a flush, which is positive edge triggered. The change state only occurs in a set that is enabled. If set is disabled, then there is nothing to flush (it is already marked as invalid or already flushed). Implemented as edge detect
9	rsvd_I	Reserved. Latch bit is implemented; value read is the value written.
10	force_geq1	Force all load instructions to be treated as if being loaded to guarded storage 0 Normal operation 1 Force all load instructions to be treated as if being loaded to guarded storage In the PPU this causes touch instructions (**dcbt**, and so on) to be treated as **nop** instructions. This may also have an affect on the ordering of stores in the CIU.
11	force_ai	Force alignment interrupt instead of microcode on unaligned operations Used for debugging. Prevents misaligned flushes. Any load or store that spans a 32-byte boundary (or an 8-byte boundary in DABR or TDABR mode) takes an alignment interrupt. 0 Normal operation 1 Force alignment interrupt.

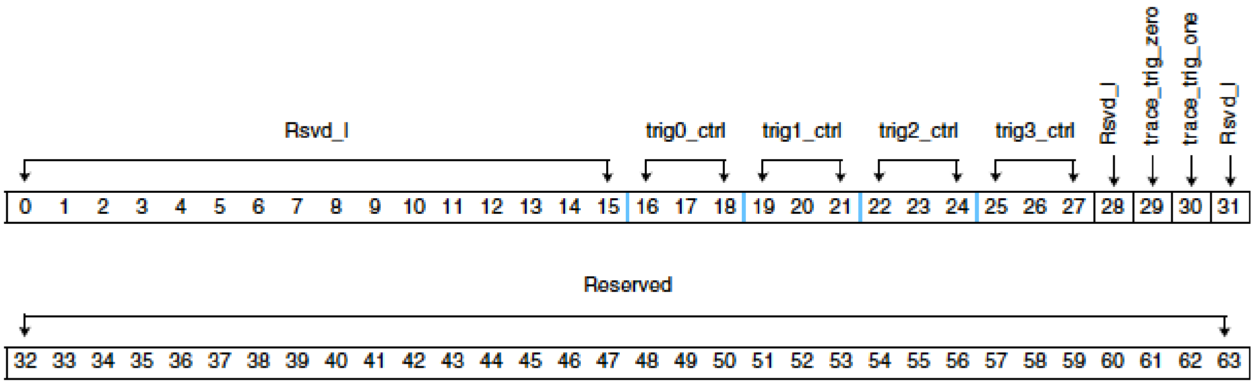
Bit(s)	Field Name	Description
12	tch_nop	Force data-cache block touch x-form (**dcbt**) and data-cache block touch for store (**dcbtst**) instructions to function like **nop** instructions. Only performs the translation. 0 Normal operation 1 Force **dcbt** and **dcbtst** to function like **nop** instructions.
13:15	lmq_size	Maximum number of outstanding demand requests to the memory subsystem (only applies to loads) ’000’ 8 outstanding requests to the PPSS ’111’ 7 outstanding requests to the PPSS ’110’ 6 outstanding requests to the PPSS ’101’ 5 outstanding requests to the PPSS ’100’ 4 outstanding requests to the PPSS ’011’ 3 outstanding requests to the PPSS ’010’ 2 outstanding requests to the PPSS ’001’ 1 outstanding requests to the PPSS
16:17	rsvd_l	Reserved. Latch bit is implemented; value read is the value written.
18	dis_force_ci	0 Force cache inhibit, unless HID4[19] disables 1 Use effective-address-to-real-address translation (ERAT) cache inhibit bit (normal translation).
19	enb_force_ci	Enable force_cache_inhibit (only valid if dis_force_ci = 0) 0 Use configuration ring (PPU/XU bit 174 (cfg_force_ci)). 1 Use HID4[18]
20:23	en_dcway	Enable L1 data cache way (one bit per way) Bit 20 enables L1 data cache way A when set to ‘1’. Bit 21 enables L1 data cache way B when set to ‘1’. Bit 22 enables L1 data cache way C when set to ‘1’. Bit 23 enables L1 data cache way D when set to ‘1’. **Note:** At POR (Power On Reset) the cache is disabled. **Note:** When all bits are zero, no writes to the L1 data cache occur. **Note:** When the L1 data cache is completely disabled, micro-coded loads and stores do not work. **Note:** Changing the value of these bits at any state other than immediately after a POR is not recommended (undefined behavior may result). If any tag in the cache has been allocated (i.e. a valid bit is on), then a flash invalidate of the tag must occur. This can be achieved by setting and then resetting HID4(8).
24:31	rsvd_l	Reserved. Latch bit is implemented; value read is the value written.
32:63	Reserved	Bits are not implemented; all bits read back zero.

Programming Note:

- After POR, this register is set to x′0000_0000_0000_0000′. In normal operation, the hypervisor should set this register to its preferred value of x′0000_3F00_0000_0000′ before booting the system.
- No changes after POR to this register are necessary in typical operating mode. The only reason to change any field of this register is for diagnostic purposes in a lab environment.

Hardware Implementation Register 5 (HID5)

Register Short Name	HID5	**Unit**	XU
Register Type	SPR Read/Write	**Width**	64 bits
Decimal SPR Number	1014	**Access Type**	Hypervisor
Value at Initial POR	All bits set to zero	**Set By**	Scan initialization during POR
Additional Information	Implementation-specific register	**Reg. Duplicated for MT**	No



Bit(s)	Field Name	Description
0:15	rsvd_l	Reserved. Latch bit is implemented; value read is the value written.
16:18	trig0_ctrl	XU Trigger 0 Select For Trace/Debug Bus '001' Select HID5[29] trace_trig_zero '010' Select HID5[30] trace_trig_one '011' Select lab dabr thread0 '100' Select lab dabr thread1 '101' select xlsu.xplb.xplb_nnn_lsu_trace_trigger (rf1_fxu_iop_v OR rf2_lsu_iop_v OR rf2_load_op OR rf2_load_or_store_op)
19:21	trig1_ctrl	XU Trigger 1 Select For Trace/Debug Bus '001' Select HID5[29] trace_trig_zero '010' Select HID5[30] trace_trig_one '011' Select lab dabr thread0 '100' Select lab dabr thread1 '101' select xlsu.xplb.xplb_nnn_lsu_trace_trigger (rf1_fxu_iop_v OR rf2_lsu_iop_v OR rf2_load_op OR rf2_load_or_store_op)
22:24	trig2_ctrl	XU Trigger 2 Select For Trace/Debug Bus '001' Select HID5[29] (trace_trig_zero) '010' Select HID5[30] (trace_trig_one) '011' Select lab dabr thread0 100' Select lab dabr thread1 '101' select xlsu.xplb.xplb_nnn_lsu_trace_trigger (rf1_fxu_iop_v OR rf2_lsu_iop_v OR rf2_load_op OR rf2_load_or_store_op)
25:27	trig3_ctrl	XU Trigger 3 Select For Trace/Debug Bus '001' Select HID5[29] trace_trig_zero '010' Select HID5[30] trace_trig_one '011' Select lab dabr thread0 '100' Select lab dabr thread1 '101' select xlsu.xplb.xplb_nnn_lsu_trace_trigger (rf1_fxu_iop_v OR rf2_lsu_iop_v OR rf2_load_op OR rf2_load_or_store_op)
28	rsvd_l	Reserved. Latch bit is implemented; value read is the value written.
29	trace_trig_zero	Special bit for triggering the trace analyzer by software. Bit is passed along on trigger bus location 0 when that trigger location is enabled for the XU.
30	trace_trig_one	Special bit for triggering the trace analyzer by software. Bit is passed along on trigger bus location one when that trigger location is enabled for XU.
31	rsvd_l	Reserved. Latch bit is implemented; value read is the value written.

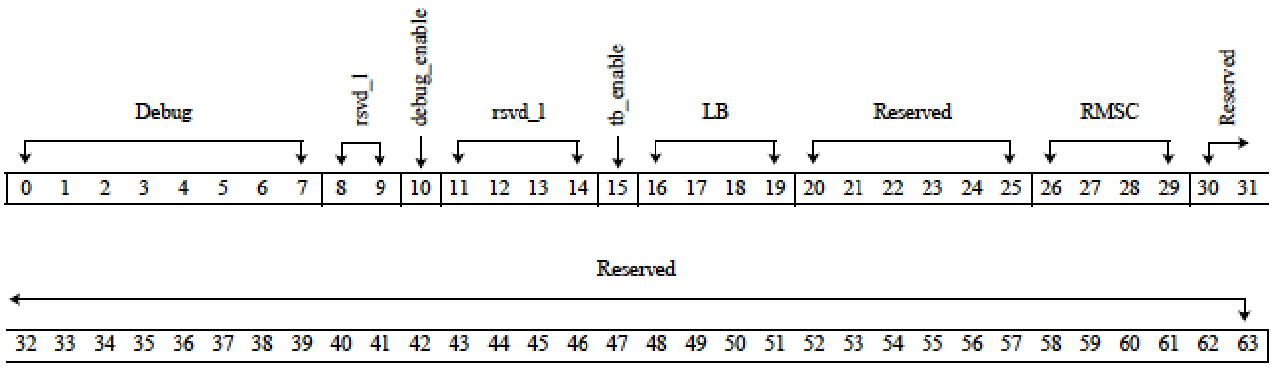
32:63	Reserved	Bits are not implemented; all bits read back zero.

Programming Note:

- After POR, this register is set to x’0000_0000_0000_0000’. In normal operation, the hypervisor should leave this register at its preferred value of x’0000_0000_0000_0000’.
- No changes after POR to this register are necessary in typical operating mode. The only reason to change any field of this register is for diagnostic purposes in a hardware-debug environment.

Hardware Implementation Register 6 (HID6)

Register Short Name	HID6	**Unit**	MMU
Register Type	SPR Read/Write	**Width**	64 bits
Decimal SPR Number	1017	**Access Type**	Hypervisor
Value at Initial POR	All bits set to zero	**Set By**	Scan initialization during POR
Additional Information	Hardware implementation register	**Reg. Duplicated for MT**	No



Bit(s)	Field Name	Description
0:7	Debug	Debug and Performance Select x'80' Default – selects performance bus x'40' MMU Control and SPR Read Debug x'20' Snoop Interface and SLB Control Debug x'10' PPU Interface Debug x'08' LRU and SPR Control Debug x'04' Time-base Debug x'02' TLB Control and Snoop Control Debug x'01' Table walk and TLB Index Debug All other combinations are not meaningful. This field is for hardware debug purposes only.
8:9	rsvd_l	Reserved. Latch bit is implemented; value read is the value written.
10	debug_enable	Turn on performance/debug bus for the MMU (see bits [0:7] for perf/debug select controls). This field is for hardware debug purposes only.
11:14	rsvd_l	Reserved. Latch bit is implemented; value read is the value written.
15	tb_enable	Time-base and decrementer facility enable 0 TBU, TBL, DEC, HDEC, and the hang-detection logic do not update (all update signals from the pervasive logic are ignored) 1 TBU, TBL, DEC, HDEC, and the hang-detection logic are enabled to update.

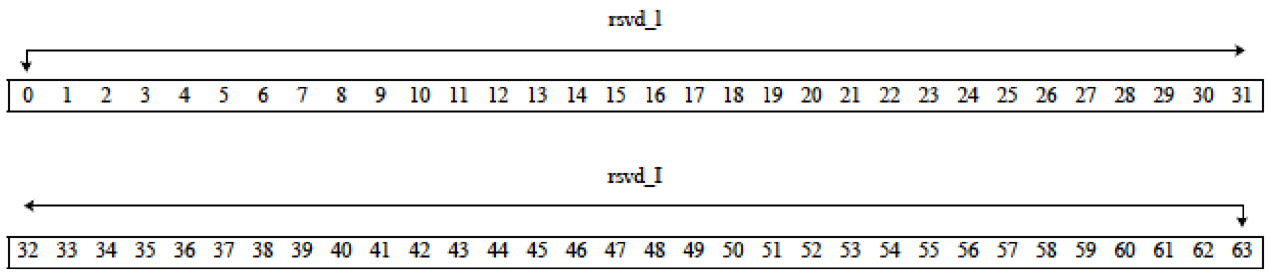
Bit(s)	Field Name	Description
16:19	LB	Large page bit table If TLB_VPN[L] = '1' and TLB_RPN[LP] = '0', then Large Page Size 1 is used. If TLB_VPN[L] = '1' and TLB_RPN[LP] = '1', then Large Page Size 2 is used. Select the concurrent large page modes. Large Page Size: Size 1 Size 2 '0000' 16 MB 16 MB '0001' 16 MB 1 MB '0010' 16 MB 64 KB '0100' 1 MB 16 MB '0101' 1 MB 1 MB '0110' 1 MB 64 KB '1000' 64 KB 16 MB '1001' 64 KB 1 MB '1010' 64 KB 64 KB All other combinations are reserved. **Additional information:** See *Section 8.4.2 Large Page-Size Decode* on page 74.
20:25	Reserved	Bits are not implemented; all bits read back zero.
26:29	RMSC	PPE real-mode storage control facility **Additional information:** See *Section 8.5.1 PPE Real-Mode Storage Control Facility* on page 79.
30:63	Reserved	Bits are not implemented; all bits read back zero.

Programming Note:

- After POR, this register is set to x'0000_0000_0000_0000'. In normal operation, the hypervisor sets HID6[tb_enable] to '1'.

Hardware Implementation Register 7 (HID7)

Register Short Name	HID7	**Unit**	IU
Register Type	SPR Read/Write	**Width**	64 bits
Decimal SPR Number	1018	**Access Type**	Hypervisor
Value at Initial POR	All bits set to zero	**Set By**	Scan initialization during POR
Additional Information	Implementation-specific register	**Reg. Duplicated for MT**	No



Bit(s)	Field Name	Description
0:63	rsvd_I	Reserved. Latch bit is implemented; value read is the value written.

Programming Note:

- This register is for hardware-debugging purposes only.

Processor Utilization of Resources Register (PURR)

This implementation does not support the PURR. Attempts to read this register return all zeros, and writes are ignored. Since the PURR is required by the architecture, this implementation is technically noncompliant. Such noncompliance is expected to have little practical implication since this is a hypervisor resource.