

e600 PowerPC™ Core

Reference Manual

E600CORERM
Rev. 0, 03/2006



How to Reach Us:

Home Page:

www.freescale.com

email:

support@freescale.com

USA/Europe or Locations Not Listed:

Freescale Semiconductor
Technical Information Center, CH370
1300 N. Alma School Road
Chandler, Arizona 85224
(800) 521-6274
480-768-2130
support@freescale.com

Europe, Middle East, and Africa:

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

Japan:

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

Asia/Pacific:

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

For Literature Requests Only:

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

Information in this document is provided solely to enable system and software implementers to use Freescale Semiconductor products. There are no express or implied copyright licenses granted hereunder to design or fabricate any integrated circuits or integrated circuits based on the information in this document.

Freescale Semiconductor reserves the right to make changes without further notice to any products herein. Freescale Semiconductor makes no warranty, representation or guarantee regarding the suitability of its products for any particular purpose, nor does Freescale Semiconductor assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation consequential or incidental damages. "Typical" parameters which may be provided in Freescale Semiconductor data sheets and/or specifications can and do vary in different applications and actual performance may vary over time. All operating parameters, including "Typicals" must be validated for each customer application by customer's technical experts. Freescale Semiconductor does not convey any license under its patent rights nor the rights of others. Freescale Semiconductor products are not designed, intended, or authorized for use as components in systems intended for surgical implant into the body, or other applications intended to support or sustain life, or for any other application in which the failure of the Freescale Semiconductor product could create a situation where personal injury or death may occur. Should Buyer purchase or use Freescale Semiconductor products for any such unintended or unauthorized application, Buyer shall indemnify and hold Freescale Semiconductor and its officers, employees, subsidiaries, affiliates, and distributors harmless against all claims, costs, damages, and expenses, and reasonable attorney fees arising out of, directly or indirectly, any claim of personal injury or death associated with such unintended or unauthorized use, even if such claim alleges that Freescale Semiconductor was negligent regarding the design or manufacture of the part.

Freescale™ and the Freescale logo are trademarks of Freescale Semiconductor, Inc. The described product is a PowerPC microprocessor core. The PowerPC name is a trademark of IBM Corp. and is used under license. All other product or service names are the property of their respective owners.

© Freescale Semiconductor, Inc., 2006. All rights reserved.



Overview	1
Registers	2
L1 and L2 Cache Operation	3
Interrupts	4
Memory Management	5
Instruction Timing	6
AltiVec Technology Implementation	7
Core Interface	8
Power and Thermal Management	9
Performance Monitor	10
e600 Core Instruction Set Listings	A
Instructions Not Implemented	B
Special-Purpose Registers	C
Revision History	D
Glossary	GLO
Index	IND



1 Overview

2 Registers

3 L1 and L2 Cache Operation

4 Interrupts

5 Memory Management

6 Instruction Timing

7 AltiVec Technology Implementation

8 Core Interface

9 Power and Thermal Management

10 Performance Monitor

A e600 Core Instruction Set Listings

B Instructions Not Implemented

C Special-Purpose Registers

D Revision History

GLO Glossary

IND Index

Contents

Paragraph Number	Title	Page Number
	Audience	xxxv
	Organization.....	xxxv
	Suggested Reading.....	xxxvi
	General Information.....	xxxvi
	Related Documentation.....	xxxvi
	Conventions	xxxvii
	Acronyms and Abbreviations	xxxvii
	Terminology Conventions.....	xli

Chapter 1 Overview

1.1	e600 Core Overview	1-1
1.2	e600 Core Features	1-4
1.2.1	Instruction Flow	1-9
1.2.1.1	Instruction Queue and Dispatch Unit	1-9
1.2.1.2	Branch Processing Unit (BPU).....	1-9
1.2.1.3	Completion Unit	1-10
1.2.1.4	Independent Execution Units.....	1-11
1.2.1.4.1	AltiVec Vector Permute Unit (VPU)	1-11
1.2.1.4.2	AltiVec Vector Integer Unit 1 (VIU1)	1-11
1.2.1.4.3	AltiVec Vector Integer Unit 2 (VIU2)	1-11
1.2.1.4.4	AltiVec Vector Floating-Point Unit (VFPU)	1-11
1.2.1.4.5	Integer Units (IUs).....	1-11
1.2.1.4.6	Floating-Point Unit (FPU).....	1-12
1.2.1.4.7	Load/Store Unit (LSU)	1-12
1.2.2	Memory Management Units (MMUs).....	1-12
1.2.3	L1 Instruction and Data Caches Within the Core	1-13
1.2.4	L2 Cache Implementation.....	1-15
1.2.5	Core Interface	1-17
1.2.6	Overview of Core Interface Accesses.....	1-17
1.2.6.1	Signal Groupings	1-18
1.2.6.2	Clocking.....	1-19
1.2.7	Power and Thermal Management	1-19
1.2.8	Core Performance Monitor	1-20
1.3	e600 Core Architectural Implementation	1-20
1.3.1	PowerPC Registers and Programming Model	1-22
1.3.2	Instruction Set	1-24

Contents

Paragraph Number	Title	Page Number
1.3.2.1	PowerPC Instruction Set.....	1-24
1.3.2.2	AltiVec Instruction Set.....	1-25
1.3.2.3	e600 Core Instruction Set	1-26
1.3.3	Cache Implementation within the Core	1-26
1.3.3.1	PowerPC Cache Model.....	1-26
1.3.3.2	e600 Core Cache Implementation	1-27
1.3.4	Interrupt Model.....	1-27
1.3.4.1	PowerPC Interrupt Model.....	1-27
1.3.4.2	e600 Core Interrupts	1-28
1.3.5	Memory Management.....	1-30
1.3.5.1	PowerPC Memory Management Model	1-30
1.3.5.2	e600 Core Memory Management Implementation.....	1-31
1.3.6	Instruction Timing	1-32
1.3.7	AltiVec Implementation.....	1-36

Chapter 2 Registers

AltiVec Technology and the Programming Model 1

2.1	e600 Core Register Set	2-1
2.1.1	Register Set Overview	2-1
2.1.2	e600 Core Register Set	2-3
2.1.3	PowerPC User-Level Registers (UISA)	2-8
2.1.4	PowerPC Supervisor-Level Registers (OEA).....	2-8
2.1.4.1	Processor Version Register (PVR).....	2-8
2.1.4.2	System Version Register (SVR).....	2-9
2.1.4.3	Processor Identification Register (PIR)	2-9
2.1.4.4	Machine State Register (MSR).....	2-9
2.1.4.5	Machine Status Save/Restore Registers (SRR0, SRR1).....	2-12
2.1.4.6	SDR1 Register	2-12
2.1.5	PowerPC User-Level Registers (VEA).....	2-13
2.1.5.1	Time Base Registers (TBL, TBU)	2-13
2.1.6	e600-Specific Register Descriptions.....	2-13
2.1.6.1	Hardware Implementation-Dependent Register 0 (HID0)	2-14
2.1.6.2	Hardware Implementation-Dependent Register 1 (HID1)	2-18
2.1.6.3	Memory Subsystem Control Register (MSSCR0).....	2-20
2.1.6.4	Memory Subsystem Status Register (MSSSR0).....	2-22
2.1.6.5	Instruction and Data Cache Registers.....	2-23
2.1.6.5.1	L2 Cache Control Register (L2CR).....	2-23
2.1.6.5.2	L2 Error Injection Mask High Register (L2ERRINJHI)	2-25
2.1.6.5.3	L2 Error Injection Mask High Register (L2ERRINJLO).....	2-25

Contents

Paragraph Number	Title	Page Number
2.1.6.5.4	L2 Error Injection Mask Control Register (L2ERRINJCTL)	2-25
2.1.6.5.5	L2 Error Capture Data High Register (L2CAPTDATAHI)	2-26
2.1.6.5.6	L2 Error Capture Data Low Register (L2CAPTDATALO).....	2-27
2.1.6.5.7	L2 Error Syndrome Register (L2CAPTECC).....	2-27
2.1.6.5.8	L2 Error Detect Register (L2ERRDET)	2-28
2.1.6.5.9	L2 Error Disable Register (L2ERRDIS)	2-29
2.1.6.5.10	L2 Error Interrupt Enable Register (L2ERRINTEN)	2-30
2.1.6.5.11	L2 Error Attributes Capture Register (L2ERRATTR)	2-30
2.1.6.5.12	L2 Error Address Error Capture Register (L2ERRADDR).....	2-31
2.1.6.5.13	L2 Error Address Error Capture Register (L2ERREADDR)	2-32
2.1.6.5.14	L2 Error Control Register (L2ERRCTL)	2-32
2.1.6.5.15	Instruction Cache and Interrupt Control Register (ICTRL)	2-33
2.1.6.5.16	Load/Store Control Register (LDSTCR)	2-34
2.1.6.6	Instruction Address Breakpoint Register (IABR).....	2-35
2.1.6.7	Memory Management Registers Used for Software Table Searching.....	2-36
2.1.6.7.1	TLB Miss Register (TLBMISS)	2-36
2.1.6.7.2	Page Table Entry Registers (PTEHI and PTELO)	2-36
2.1.6.8	Thermal Management Register.....	2-38
2.1.6.8.1	Instruction Cache Throttling Control Register (ICTC)	2-38
2.1.6.9	Performance Monitor Registers	2-39
2.1.6.9.1	Monitor Mode Control Register 0 (MMCR0)	2-39
2.1.6.9.2	User Monitor Mode Control Register 0 (UMMCR0).....	2-41
2.1.6.9.3	Monitor Mode Control Register 1 (MMCR1)	2-42
2.1.6.9.4	User Monitor Mode Control Register 1 (UMMCR1).....	2-42
2.1.6.9.5	Monitor Mode Control Register 2 (MMCR2)	2-42
2.1.6.9.6	User Monitor Mode Control Register 2 (UMMCR2).....	2-43
2.1.6.9.7	Breakpoint Address Mask Register (BAMR).....	2-43
2.1.6.9.8	Performance Monitor Counter Registers (PMC1–PMC6)	2-44
2.1.6.9.9	User Performance Monitor Counter Registers (UPMC1–UPMC6).....	2-45
2.1.6.9.10	Sampled Instruction Address Register (SIAR).....	2-45
2.1.6.9.11	User-Sampled Instruction Address Register (USIAR).....	2-45
2.1.6.9.12	Sampled Data Address Register (SDAR) and User-Sampled Data Address Register (USDAR)	2-45
2.1.7	Reset Settings.....	2-45
2.2	Operand Conventions	2-48
2.2.1	Floating-Point Execution Models—UISA.....	2-48
2.2.2	Data Organization in Memory and Data Transfers.....	2-48
2.2.3	Alignment and Misaligned Accesses.....	2-49
2.2.4	Floating-Point Operands.....	2-49
2.3	Instruction Set Summary	2-50
2.3.1	Classes of Instructions	2-51

Contents

Paragraph Number	Title	Page Number
2.3.1.1	Definition of Boundedly Undefined	2-51
2.3.1.2	Defined Instruction Class	2-51
2.3.1.3	Illegal Instruction Class	2-52
2.3.1.4	Reserved Instruction Class	2-53
2.3.2	Addressing Modes	2-53
2.3.2.1	Memory Addressing	2-53
2.3.2.2	Memory Operands	2-53
2.3.2.3	Effective Address Calculation	2-54
2.3.2.4	Synchronization	2-54
2.3.2.4.1	Context Synchronization	2-54
2.3.2.4.2	Execution Synchronization	2-58
2.3.2.4.3	Instruction-Related Interrupts	2-58
2.3.3	Instruction Set Overview	2-58
2.3.4	PowerPC UIA Instructions	2-59
2.3.4.1	Integer Instructions	2-59
2.3.4.1.1	Integer Arithmetic Instructions	2-59
2.3.4.1.2	Integer Compare Instructions	2-60
2.3.4.1.3	Integer Logical Instructions	2-61
2.3.4.1.4	Integer Rotate and Shift Instructions	2-61
2.3.4.2	Floating-Point Instructions	2-62
2.3.4.2.1	Floating-Point Arithmetic Instructions	2-63
2.3.4.2.2	Floating-Point Multiply-Add Instructions	2-63
2.3.4.2.3	Floating-Point Rounding and Conversion Instructions	2-64
2.3.4.2.4	Floating-Point Compare Instructions	2-64
2.3.4.2.5	Floating-Point Status and Control Register Instructions	2-64
2.3.4.2.6	Floating-Point Move Instructions	2-65
2.3.4.3	Load and Store Instructions	2-65
2.3.4.3.1	Self-Modifying Code	2-66
2.3.4.3.2	Integer Load and Store Address Generation	2-66
2.3.4.3.3	Register Indirect Integer Load Instructions	2-66
2.3.4.3.4	Integer Store Instructions	2-68
2.3.4.3.5	Integer Store Gathering	2-68
2.3.4.3.6	Integer Load and Store with Byte-Reverse Instructions	2-69
2.3.4.3.7	Integer Load and Store Multiple Instructions	2-69
2.3.4.3.8	Integer Load and Store String Instructions	2-69
2.3.4.3.9	Floating-Point Load and Store Address Generation	2-70
2.3.4.3.10	Floating-Point Store Instructions	2-71
2.3.4.4	Branch and Flow Control Instructions	2-73
2.3.4.4.1	Branch Instruction Address Calculation	2-73
2.3.4.4.2	Branch Instructions	2-73
2.3.4.4.3	Condition Register Logical Instructions	2-74

Contents

Paragraph Number	Title	Page Number
2.3.4.4.4	Trap Instructions	2-74
2.3.4.5	System Linkage Instruction—UISA	2-74
2.3.4.6	Processor Control Instructions—UISA	2-75
2.3.4.6.1	Move To/From Condition Register Instructions	2-75
2.3.4.6.2	Move To/From Special-Purpose Register Instructions (UISA)	2-75
2.3.4.7	Memory Synchronization Instructions—UISA	2-77
2.3.5	PowerPC VEA Instructions	2-77
2.3.5.1	Processor Control Instructions—VEA	2-78
2.3.5.2	Memory Synchronization Instructions—VEA	2-78
2.3.5.3	Memory Control Instructions—VEA	2-79
2.3.5.3.1	User-Level Cache Instructions—VEA	2-79
2.3.5.4	Optional External Control Instructions	2-81
2.3.6	PowerPC OEA Instructions	2-82
2.3.6.1	System Linkage Instructions—OEA	2-82
2.3.6.2	Processor Control Instructions—OEA	2-82
2.3.6.3	Memory Control Instructions—OEA	2-86
2.3.6.3.1	Supervisor-Level Cache Management Instruction—(OEA)	2-87
2.3.6.3.2	Translation Lookaside Buffer Management Instructions—OEA	2-87
2.3.7	Recommended Simplified Mnemonics	2-88
2.3.8	Implementation-Specific Instructions	2-88
2.4	Altivec Instructions	2-91
2.5	Altivec UISA Instructions	2-92
2.5.1	Vector Integer Instructions	2-92
2.5.1.1	Vector Integer Arithmetic Instructions	2-92
2.5.1.2	Vector Integer Compare Instructions	2-94
2.5.1.3	Vector Integer Logical Instructions	2-95
2.5.1.4	Vector Integer Rotate and Shift Instructions	2-95
2.5.2	Vector Floating-Point Instructions	2-95
2.5.2.1	Vector Floating-Point Arithmetic Instructions	2-96
2.5.2.2	Vector Floating-Point Multiply-Add Instructions	2-96
2.5.2.3	Vector Floating-Point Rounding and Conversion Instructions	2-96
2.5.2.4	Vector Floating-Point Compare Instructions	2-97
2.5.2.5	Vector Floating-Point Estimate Instructions	2-97
2.5.3	Vector Load and Store Instructions	2-98
2.5.3.1	Vector Load Instructions	2-98
2.5.3.2	Vector Load Instructions Supporting Alignment	2-98
2.5.3.3	Vector Store Instructions	2-99
2.5.4	Control Flow	2-99
2.5.5	Vector Permutation and Formatting Instructions	2-99
2.5.5.1	Vector Pack Instructions	2-99
2.5.5.2	Vector Unpack Instructions	2-100

Contents

Paragraph Number	Title	Page Number
2.5.5.3	Vector Merge Instructions.....	2-100
2.5.5.4	Vector Splat Instructions.....	2-101
2.5.5.5	Vector Permute Instructions.....	2-101
2.5.5.6	Vector Select Instruction.....	2-102
2.5.5.7	Vector Shift Instructions	2-102
2.5.5.8	Vector Status and Control Register Instructions.....	2-102
2.6	Altivec VEA Instructions	2-103
2.6.1	Altivec Vector Memory Control Instructions—VEA.....	2-103
2.6.2	Altivec Instructions with Specific Implementations for the e600 Core	2-104

Chapter 3 L1 and L2 Cache Operation

3.1	Overview.....	3-1
3.1.1	Block Diagram.....	3-4
3.1.2	Load/Store Unit (LSU)	3-6
3.1.2.1	Cacheable Loads and LSU.....	3-6
3.1.2.2	LSU Store Queues	3-6
3.1.2.3	Store Gathering/Merging	3-7
3.1.2.4	LSU Load Miss, Castout, and Push Queues.....	3-7
3.1.3	Core Memory Subsystem Blocks	3-8
3.1.3.1	L1 Service Queues.....	3-8
3.1.3.2	L2 Cache Block	3-9
3.1.3.3	Core Interface Block.....	3-9
3.2	L1 Cache Organizations.....	3-9
3.2.1	L1 Data Cache Organization.....	3-10
3.2.2	L1 Instruction Cache Organization.....	3-11
3.3	Memory and Cache Coherency.....	3-12
3.3.1	Memory/Cache Access Attributes (WIMG Bits).....	3-12
3.3.1.1	Coherency Paradoxes and WIMG	3-13
3.3.1.2	Out-of-Order Accesses to Guarded Memory.....	3-13
3.3.2	Coherency Support	3-14
3.3.2.1	Coherency Between L1 and L2 Caches.....	3-15
3.3.2.1.1	Cache Closer to Core with Modified Data	3-16
3.3.2.1.2	Transient Data and the L2 Cache.....	3-16
3.3.2.2	Snoop Response.....	3-16
3.3.2.3	Intervention.....	3-17
3.3.2.4	Simplified Transaction Types	3-17
3.3.2.5	MESI State Transitions	3-18
3.3.2.5.1	MESI Protocol with Data Intervention Enabled.....	3-18
3.3.2.5.2	MESI Protocol (with Intervention Disabled).....	3-21

Contents

Paragraph Number	Title	Page Number
3.3.2.6	Reservation Snooping	3-23
3.3.3	Load/Store Operations and Architecture Implications	3-24
3.3.3.1	Performed Loads and Store.....	3-24
3.3.3.2	Sequential Consistency of Memory Accesses	3-25
3.3.3.3	Load Ordering with Respect to Other Loads	3-26
3.3.3.4	Store Ordering with Respect to Other Stores.....	3-26
3.3.3.5	Enforcing Store Ordering with Respect to Loads	3-26
3.3.3.6	Atomic Memory References	3-26
3.4	L1 Cache Control.....	3-27
3.4.1	Cache Control Parameters in HID0	3-28
3.4.1.1	Enabling and Disabling the Data Cache	3-28
3.4.1.2	Data Cache Locking with DLOCK.....	3-28
3.4.1.3	Enabling and Disabling the Instruction Cache	3-29
3.4.1.4	Instruction Cache Locking with ILOCK	3-29
3.4.1.5	L1 Instruction and Data Cache Flash Invalidation	3-29
3.4.2	Data Cache Way Locking Setting in LDSTCR	3-30
3.4.3	Cache Control Parameters in ICTRL.....	3-30
3.4.3.1	Instruction Cache Way Locking	3-30
3.4.3.2	Enabling Instruction Cache Parity Checking.....	3-31
3.4.3.3	Instruction and Data Cache Parity Error Reporting.....	3-31
3.4.4	Cache Control Instructions	3-31
3.4.4.1	Data Cache Block Touch (dcbt)	3-31
3.4.4.2	Data Cache Block Touch for Store (dcbtst)	3-32
3.4.4.3	Data Cache Block Zero (dcbz)	3-33
3.4.4.4	Data Cache Block Store (dcbst)	3-33
3.4.4.5	Data Cache Block Flush (dcbf)	3-34
3.4.4.6	Data Cache Block Allocate (dcba).....	3-34
3.4.4.7	Data Cache Block Invalidate (dcbi)	3-35
3.4.4.8	Instruction Cache Block Invalidate (icbi).....	3-35
3.5	L1 Cache Operation	3-36
3.5.1	Cache Miss and Reload Operations	3-36
3.5.1.1	Data Cache Fills.....	3-36
3.5.1.2	Instruction Cache Fills.....	3-36
3.5.2	Cache Allocation on Misses	3-37
3.5.2.1	Instruction Access Allocation in L1 Cache	3-37
3.5.2.2	Data Access Allocation in L1 Cache	3-37
3.5.3	Store Miss Merging.....	3-38
3.5.4	Load/Store Miss Handling	3-38
3.5.5	Store Hit to a Data Cache Block Marked Shared	3-38
3.5.6	Data Cache Block Push Operation.....	3-38
3.5.7	L1 Cache Block Replacement Selection.....	3-38

Contents

Paragraph Number	Title	Page Number
3.5.7.1	PLRU Replacement	3-38
3.5.7.2	PLRU Bit Updates	3-39
3.5.7.3	AltiVec LRU Instruction Support	3-40
3.5.7.4	Cache Locking and PLRU	3-41
3.5.8	L1 Cache Invalidation and Flushing	3-41
3.5.9	L1 Cache Operation Summary	3-42
3.6	L2 Cache	3-45
3.6.1	L2 Cache Organization	3-45
3.6.2	L2 Cache and Memory Coherency	3-46
3.6.3	L2 Cache Control	3-46
3.6.3.1	L2CR Parameters	3-46
3.6.3.1.1	Enabling the L2 Cache and L2 Initialization	3-46
3.6.3.1.2	Enabling L2 Parity Checking	3-47
3.6.3.1.3	L2 Instruction-Only and Data-Only Modes	3-47
3.6.3.1.4	L2 Cache Invalidation	3-47
3.6.3.1.5	Flushing of L1 and L2 Caches	3-48
3.6.3.1.6	L2 Replacement Algorithm Selection	3-49
3.6.3.2	L2 Prefetch Engines and MSSCR0	3-49
3.6.3.3	L2 Parity Error Reporting	3-49
3.6.3.4	L2 Data ECC	3-49
3.6.3.4.1	Enabling or Disabling ECC	3-50
3.6.3.4.2	L2 Error Control and Capture	3-50
3.6.3.4.3	ECC Error Reporting	3-51
3.6.3.4.4	L2 Error Injection	3-51
3.6.3.5	Instruction Interactions with L2	3-51
3.6.4	L2 Cache Operation	3-52
3.6.4.1	L2 Cache Miss and Reload Operations	3-53
3.6.4.2	L2 Cache Allocation	3-53
3.6.4.3	Store Data Merging and L2	3-54
3.6.4.4	L2 Cache Line Replacement Algorithms	3-54
3.6.4.5	L2 Operations Caused by L1 Requests	3-55
3.7	Core Interface	3-58
3.7.1	MPX bus Operations Caused by Cache Control Instructions	3-59
3.7.2	Transfer Attributes	3-60
3.7.3	Snooping of MPX Bus Transactions	3-62
3.7.3.1	Types of Transactions Snooped by the e600 Core	3-62
3.7.3.2	L1 Cache State Transitions and Bus Operations Due to Snoops	3-64
3.7.3.3	L2 Operations Caused by Snoops External to the Core	3-65

Contents

Paragraph Number	Title	Page Number
Chapter 4		
Interrupts		
4.1	e600 Core Interrupts	4-3
4.2	e600 Core Interrupt Recognition and Priorities	4-5
4.3	Interrupt Processing	4-8
4.3.1	Enabling and Disabling Exceptions and Interrupts.....	4-12
4.3.2	Steps for Interrupt Processing.....	4-12
4.3.3	Setting MSR[RI].....	4-13
4.3.4	Returning from an Interrupt Handler	4-13
4.4	Process Switching	4-13
4.5	Data Stream Prefetching and Interrupts	4-14
4.6	Interrupt Definitions	4-14
4.6.1	System Reset Interrupt (0x00100)	4-15
4.6.2	Machine Check Interrupt (0x00200).....	4-16
4.6.2.1	Machine Check Interrupt Enabled (MSR[ME] = 1).....	4-18
4.6.2.2	Checkstop State (MSR[ME] = 0)	4-20
4.6.3	DSI Interrupt (0x00300)	4-20
4.6.3.1	DSI Interrupt—Page Fault	4-20
4.6.3.2	DSI Interrupt—Data Address Breakpoint Facility	4-21
4.6.4	ISI Interrupt (0x00400).....	4-21
4.6.5	External Interrupt (0x00500)	4-22
4.6.6	Alignment Interrupt (0x00600).....	4-23
4.6.7	Program Interrupt (0x00700)	4-23
4.6.8	Floating-Point Unavailable Interrupt (0x00800)	4-24
4.6.9	Decrementer Interrupt (0x00900)	4-24
4.6.10	System Call Interrupt (0x00C00).....	4-24
4.6.11	Trace Interrupt (0x00D00)	4-24
4.6.12	Floating-Point Assist Interrupt (0x00E00)	4-25
4.6.13	Performance Monitor Interrupt (0x00F00)	4-25
4.6.14	AltiVec Unavailable Interrupt (0x00F20).....	4-26
4.6.15	TLB Miss Interrupts.....	4-26
4.6.15.1	Instruction Table Miss Interrupt—ITLB Miss (0x01000)	4-27
4.6.15.2	Data Table Miss-on-Load Interrupt—DTLB Miss-on-Load (0x01100).....	4-28
4.6.15.3	Data Table Miss-on-Store Interrupt—DTLB Miss-on-Store (0x01200)	4-28
4.6.16	Instruction Address Breakpoint Interrupt (0x01300).....	4-28
4.6.17	System Management Interrupt (0x01400)	4-29
4.6.18	AltiVec Assist Interrupt (0x01600).....	4-30

Contents

Paragraph Number	Title	Page Number
Chapter 5		
Memory Management		
5.1	MMU Overview	5-2
5.1.1	Memory Addressing	5-4
5.1.2	MMU Organization.....	5-4
5.1.3	Address Translation Mechanisms	5-9
5.1.4	Memory Protection Facilities.....	5-13
5.1.5	Page History Information.....	5-13
5.1.6	General Flow of MMU Address Translation	5-14
5.1.6.1	Real Addressing Mode and Block Address Translation Selection	5-14
5.1.6.2	Page Address Translation Selection	5-15
5.1.7	MMU Interrupts Summary	5-18
5.1.8	MMU Instructions and Register Summary	5-20
5.2	Real Addressing Mode.....	5-22
5.2.1	Real Addressing Mode—32-Bit Addressing	5-23
5.2.2	Real Addressing Mode—Extended Addressing	5-23
5.3	Block Address Translation.....	5-23
5.3.1	BAT Register Implementation of BAT Array—Extended Addressing.....	5-24
5.3.2	Block Physical Address Generation—Extended Addressing	5-26
5.3.2.1	Block Physical Address Generation with an Extended BAT Block Size	5-27
5.3.3	Block Address Translation Summary—Extended Addressing.....	5-29
5.4	Memory Segment Model	5-31
5.4.1	Page Address Translation Overview.....	5-32
5.4.1.1	Segment Descriptor Definitions	5-33
5.4.1.2	Page Table Entry (PTE) Definition—Extended Addressing.....	5-34
5.4.2	Page History Recording	5-35
5.4.2.1	Referenced Bit	5-36
5.4.2.2	Changed Bit	5-36
5.4.2.3	Scenarios for Referenced and Changed Bit Recording	5-37
5.4.3	Page Memory Protection	5-38
5.4.4	TLB Description	5-38
5.4.4.1	TLB Organization and Operation	5-38
5.4.4.2	TLB Invalidation	5-40
5.4.4.2.1	tlbie Instruction	5-40
5.4.4.2.2	tlbsync Instruction.....	5-42
5.4.4.2.3	Synchronization Requirements for tlbie and tlbsync	5-43
5.4.5	Page Address Translation Summary—Extended Addressing.....	5-44
5.5	Hashed Page Tables—Extended Addressing	5-46
5.5.1	SDR1 Register Definition—Extended Addressing.....	5-46
5.5.1.1	Page Table Size.....	5-48

Contents

Paragraph Number	Title	Page Number
5.5.1.2	Page Table Hashing Functions.....	5-49
5.5.1.3	Page Table Address Generation.....	5-50
5.5.1.4	Page Table Structure Example—Extended Addressing.....	5-52
5.5.1.5	PTEG Address Mapping Examples—Extended Addressing	5-53
5.5.2	Page Table Search Operations—Implementation	5-56
5.5.2.1	Conditions for a Page Table Search Operation.....	5-56
5.5.2.2	Altivec Line Fetch Skipping	5-56
5.5.2.3	Page Table Search Operation—Conceptual Flow	5-57
5.5.3	Page Table Updates.....	5-60
5.5.4	Segment Register Updates	5-61
5.5.5	Implementation-Specific Software Table Search Operation	5-61
5.5.5.1	Resources for Table Search Operations.....	5-61
5.5.5.1.1	TLB Miss Register (TLBMISS).....	5-63
5.5.5.1.2	Page Table Entry Registers (PTEHI and PTELO)	5-64
5.5.5.1.3	Special Purpose Registers (4–7).....	5-65
5.5.5.2	Example Software Table Search Operation	5-65
5.5.5.2.1	Flow for Example Interrupt Handlers.....	5-66
5.5.5.2.2	Code for Example Interrupt Handlers	5-71

Chapter 6 Instruction Timing

6.1	Terminology and Conventions.....	6-1
6.2	Instruction Timing Overview.....	6-3
6.3	Timing Considerations.....	6-9
6.3.1	General Instruction Flow	6-10
6.3.2	Instruction Fetch Timing.....	6-15
6.3.2.1	Cache Arbitration.....	6-15
6.3.2.2	Cache Hit	6-15
6.3.2.3	Cache Miss.....	6-18
6.3.2.4	L2 Cache Access Timing Considerations	6-20
6.3.2.4.1	Instruction Cache Miss/L2 Cache Hit	6-20
6.3.3	Dispatch, Issue, and Completion Considerations	6-22
6.3.3.1	Rename Register Operation	6-22
6.3.3.2	Instruction Serialization.....	6-23
6.4	Execution Unit Timings.....	6-23
6.4.1	Branch Processing Unit Execution Timing.....	6-23
6.4.1.1	Branch Folding and Removal of Fall-Through Branch Instructions	6-24
6.4.1.2	Branch Instructions and Completion	6-25
6.4.1.3	Branch Prediction and Resolution	6-26
6.4.1.3.1	Static Branch Prediction	6-27

Contents

Paragraph Number	Title	Page Number
6.4.1.3.2	Predicted Branch Timing Examples	6-28
6.4.2	Integer Unit Execution Timing	6-30
6.4.3	FPU Execution Timing	6-30
6.4.3.1	Effect of Floating-Point Exceptions on Performance	6-31
6.4.4	Load/Store Unit Execution Timing	6-31
6.4.4.1	Effect of Operand Placement on Performance	6-31
6.4.4.2	Store Gathering	6-32
6.4.4.3	AltiVec Instructions Executed by the LSU	6-33
6.4.4.3.1	LRU Instructions	6-33
6.4.4.3.2	Transient Instructions	6-33
6.4.5	AltiVec Instructions	6-33
6.4.5.1	AltiVec Unit Execution Timing	6-34
6.4.5.1.1	AltiVec Permute Unit (VPU) Execution Timing	6-34
6.4.5.1.2	Vector Simple Integer Unit (VIU1) Execution Timing	6-34
6.4.5.1.3	Vector Complex Integer Unit (VIU2) Execution Timing	6-34
6.4.5.1.4	Vector Floating-Point Unit (VFPU) Execution Timing	6-34
6.5	Memory Performance Considerations	6-36
6.5.1	Caching and Memory Coherency	6-36
6.6	Instruction Latency Summary	6-37
6.7	Instruction Scheduling Guidelines	6-49
6.7.1	Fetch/Branch Considerations	6-50
6.7.1.1	Fetching Examples	6-50
6.7.1.1.1	Fetch Alignment Example	6-51
6.7.1.1.2	Branch-Taken Bubble Example	6-52
6.7.1.2	Branch Conditionals	6-53
6.7.1.2.1	Branch Mispredict Example	6-53
6.7.1.2.2	Branch Loop Example	6-53
6.7.1.3	Static versus Dynamic Prediction	6-55
6.7.1.4	Using the Link Stack for Branch Indirect	6-56
6.7.1.4.1	Link Stack Example	6-56
6.7.1.4.2	Position-Independent Code Example	6-57
6.7.1.5	Branch Folding	6-58
6.7.2	Dispatch Unit Resource Requirements	6-58
6.7.2.1	Dispatch Groupings	6-59
6.7.2.1.1	Dispatch Stall Due to Rename Availability	6-59
6.7.2.2	Dispatching Load/Store Strings and Multiples	6-60
6.7.2.2.1	Example of Load/Store Multiple Micro-Operation Generation	6-60
6.7.3	Issue Queue Resource Requirements	6-61
6.7.3.1	GPR Issue Queue (GIQ)	6-61
6.7.3.2	Vector Issue Queue (VIQ)	6-62
6.7.3.3	Floating-Point Issue Queue (FIQ)	6-63

Contents

Paragraph Number	Title	Page Number
6.7.4	Completion Unit Resource Requirements	6-63
6.7.4.1	Completion Groupings.....	6-63
6.7.5	Serialization Effects	6-63
6.7.6	Execution Unit Considerations	6-64
6.7.6.1	IU1 Considerations	6-64
6.7.6.2	IU2 Considerations	6-65
6.7.6.3	FPU Considerations	6-65
6.7.6.4	Vector Unit Considerations	6-67
6.7.6.5	Load/Store Unit (LSU)	6-68
6.7.6.5.1	Load Hit Pipeline.....	6-69
6.7.6.5.2	Store Hit Pipeline.....	6-70
6.7.6.5.3	Load/Store Interaction	6-71
6.7.6.5.4	Misalignment Effects.....	6-72
6.7.6.5.5	Load Miss Pipeline	6-72
6.7.6.5.6	Store Miss Pipeline	6-74
6.7.6.5.7	DST Instructions and the Vector Touch Engine (VTE).....	6-76
6.7.7	Core Memory Subsystem Considerations.....	6-77
6.7.7.1	L2 Cache Effects.....	6-77
6.7.7.2	Hardware Prefetching	6-77

Chapter 7 AltiVec Technology Implementation

7.1	AltiVec Technology and the Programming Model	7-1
7.1.1	Register Set	7-2
7.1.1.1	Changes to the Condition Register	7-2
7.1.1.2	Addition to the Machine State Register	7-2
7.1.1.3	Vector Registers (VRs)	7-2
7.1.1.4	Vector Status and Control Register (VSCR).....	7-3
7.1.1.5	Vector Save/Restore Register (VRSAVE)	7-4
7.1.2	AltiVec Instruction Set.....	7-4
7.1.2.1	LRU Instructions	7-4
7.1.2.2	Transient Instructions and Caches	7-5
7.1.2.3	Data Stream Touch Instructions.....	7-6
7.1.2.3.1	Stream Engine Tags	7-7
7.1.2.3.2	Speculative Execution and Pipeline Stalls for Data Stream Instructions.....	7-7
7.1.2.3.3	Static/Transient Data Stream Touch Instructions	7-8
7.1.2.3.4	Relationship with the sync/tblsync Instructions	7-8
7.1.2.3.5	Data Stream Termination	7-8
7.1.2.3.6	Line Fetch Skipping.....	7-9

Contents

Paragraph Number	Title	Page Number
7.1.2.3.7	Context Awareness and Stream Pausing.....	7-9
7.1.2.3.8	Differences Between dst/dstt and dstst/dststt Instructions.....	7-9
7.1.2.4	dss and dssall Instructions.....	7-10
7.1.2.5	Java Mode, NaNs, Denormalized Numbers, and Zeros.....	7-10
7.1.2.6	AltiVec Instructions with Unique Behaviors.....	7-14
7.1.2.7	AltiVec Instruction Sequencing.....	7-14
7.2	AltiVec Technology and the Cache Model.....	7-15
7.3	AltiVec and the Interrupt Model.....	7-15
7.4	AltiVec and the Memory Management Model.....	7-16
7.5	AltiVec Technology and Instruction Timing.....	7-16

Chapter 8 Core Interface

8.1	Signal Overview.....	8-1
8.1.1	Signal Descriptions.....	8-2
8.1.2	Configuration Signals Sampled at Reset.....	8-4
8.1.3	Reset, Interrupt, Checkstop, and Power Management Signal Interactions.....	8-5
8.1.3.1	Reset Inputs.....	8-5
8.1.3.2	External Interrupts.....	8-5
8.1.3.3	Checkstops.....	8-5
8.1.3.4	Power Management Signals.....	8-5
8.1.4	IEEE Std. 1149.1a-1993 Interface.....	8-6
8.1.4.1	JTAG/COP Interface.....	8-6
8.2	e600 Core Interface Summary.....	8-7
8.2.1	MPX Bus Features.....	8-7
8.2.2	Overview of Core Interface Accesses.....	8-7
8.2.3	Summary of L1 Instruction and Data Cache Operation.....	8-9
8.2.4	L2 Cache Overview.....	8-10
8.2.5	Operation of the Core Interface.....	8-10
8.2.6	Memory Subsystem Control Register (MSSCR0).....	8-10
8.2.7	Memory Subsystem Status Register (MSSSR0).....	8-10
8.2.8	Direct-Store Accesses Not Supported.....	8-11
8.2.8.1	Address Transfer Attributes.....	8-11
8.2.8.1.1	Transfer Type (<i>tt</i> [0:4]) Signals.....	8-11
8.2.9	Write-Through (<i>wt</i>), Cache Inhibit (<i>ci</i>), and Global (<i>gbl</i>) Signals.....	8-12

Contents

Paragraph Number	Title	Page Number
Chapter 9		
Power and Thermal Management		
9.1	Overview	9-1
9.2	Power Management Signals.....	9-1
9.3	Core Power Management States	9-2
9.3.1	Full-Power State	9-3
9.3.2	Doze State	9-3
9.3.2.1	Entering Doze State	9-3
9.3.2.2	Exiting Doze State	9-4
9.3.3	Nap State.....	9-4
9.3.3.1	Entering Nap State	9-4
9.3.3.2	Exiting Nap State	9-4
9.3.3.3	Snooping in Nap State	9-4
9.3.4	Sleep State.....	9-4
9.3.4.1	Entering Sleep State.....	9-5
9.3.4.2	Exiting Sleep State.....	9-5
9.3.4.3	Snooping in Sleep State	9-5
9.3.5	Deep Sleep State	9-5
9.3.5.1	Entering Deep Sleep State	9-5
9.3.5.2	Exiting Deep Sleep State	9-5
9.3.6	Power Management Software Considerations	9-5
9.4	Power Management Control Bits.....	9-6
9.5	Power Management Protocol.....	9-7
9.6	Interrupts and Power Management	9-8
9.6.1	Dynamic Frequency Switching (DFS).....	9-8
9.6.1.1	Snooping Restrictions	9-8
9.7	Instruction Cache Throttling.....	9-9

Chapter 10 Performance Monitor

10.1	Overview.....	10-2
10.2	Performance Monitor Interrupt.....	10-2
10.2.1	Performance Monitor Signals	10-3
10.2.2	Using Timebase Event to Trigger or Freeze a Counter or Generate an Interrupt	10-3
10.3	Performance Monitor Registers	10-3
10.3.1	Performance Monitor Special-Purpose Registers.....	10-3
10.3.2	Monitor Mode Control Register 0 (MMCR0)	10-4
10.3.2.1	User Monitor Mode Control Register 0 (UMMCR0).....	10-7
10.3.3	Monitor Mode Control Register 1 (MMCR1)	10-8

Contents

Paragraph Number	Title	Page Number
10.3.3.1	User Monitor Mode Control Register 1 (UMMCR1).....	10-8
10.3.4	Monitor Mode Control Register 2 (MMCR2)	10-8
10.3.4.1	User Monitor Mode Control Register 2 (UMMCR2).....	10-9
10.3.5	Breakpoint Address Mask Register (BAMR).....	10-9
10.3.6	Performance Monitor Counter Registers (PMC1–PMC6).....	10-10
10.3.6.1	User Performance Monitor Counter Registers (UPMC1–UPMC6)	10-11
10.3.7	Sampled Instruction Address Register (SIAR).....	10-11
10.3.7.1	User Sampled Instruction Address Register (USIAR)	10-12
10.4	Event Counting	10-12
10.5	Event Selection	10-12
10.5.1	PMC1 Events	10-13
10.5.2	PMC2 Events	10-19
10.5.3	PMC3 Events	10-23
10.5.4	PMC4 Events	10-25
10.5.5	PMC5 Events	10-27
10.5.6	PMC6 Events	10-28

Appendix A e600 Core Instruction Set Listings

A.1	Instructions Sorted by Mnemonic (Decimal and Hexadecimal).....	A-1
A.2	Instructions Sorted by Primary and Secondary Opcodes (Decimal and Hexadecimal).....	A-12
A.3	Instructions Sorted by Mnemonic (Binary).....	A-23
A.4	Instructions Sorted by Opcode (Binary).....	A-34
A.5	Instructions Grouped by Functional Categories.....	A-45
A.6	Instructions Sorted by Form.....	A-59
A.7	Instruction Set Legend.....	A-75

Appendix B Instructions Not Implemented

Appendix C Special-Purpose Registers

Appendix D Revision History

Glossary

Index

Figures

Figure Number	Title	Page Number
1-1	e600 Core Block Diagram.....	1-3
1-2	L1 Cache Organization	1-14
1-3	Alignment of Target Instructions in the BTIC	1-15
1-4	L2 Cache Organization	1-16
1-5	Core Interface Signals.....	1-19
1-6	Programming Model—e600 Core Registers.....	1-23
1-7	Pipelined Execution Unit	1-33
1-8	Superscalar/Pipeline Diagram.....	1-34
2-1	Programming Model—e600 Core Registers.....	2-2
2-2	Machine State Register (MSR)	2-9
2-3	Machine Status Save/Restore Register 0 (SRR0)	2-12
2-4	Machine Status Save/Restore Register 1 (SRR1)	2-12
2-5	SDR1 Register Format—Extended Addressing.....	2-13
2-6	Hardware Implementation-Dependent Register 0 (HID0).....	2-14
2-7	Hardware Implementation-Dependent Register 1 (HID1).....	2-19
2-8	Memory Subsystem Control Register (MSSCR0).....	2-21
2-9	MSS Status Register (MSSSR0)	2-22
2-10	L2 Control Register (L2CR)	2-23
2-11	L2 Error Injection Mask High Register (L2ERRINJHI)	2-25
2-12	L2 Error Injection Mask Low Register (L2ERRINJLO).....	2-25
2-13	L2 Error Injection Mask Control Register (L2ERRINJCTL).....	2-26
2-14	L2 Error Capture Data High Register (L2CAPTDATAHI).....	2-26
2-15	L2 Error Capture Data Low Register (L2CAPTDATALO)	2-27
2-16	L2 Error Syndrome Register (L2CAPTECC)	2-27
2-17	L2 Error Detect Register (L2ERRDET)	2-28
2-18	L2 Error Disable Register (L2ERRDIS).....	2-29
2-19	L2 Error Interrupt Enable Register (L2ERRINTEN)	2-30
2-20	L2 Error Attributes Capture Register (L2ERRATTR).....	2-30
2-21	L2 Error Address Error Capture Register (L2ERRADDR)	2-31
2-22	L2 Error Address Error Capture Register (L2ERRREADDR).....	2-32
2-23	L2 Error Control Register (L2ERRCTL).....	2-32
2-24	Instruction Cache and Interrupt Control Register (ICTRL).....	2-33
2-25	Load/Store Control Register (LDSTCR)	2-34
2-26	Instruction Address Breakpoint Register (IABR).....	2-35
2-27	TLBMISS Register	2-36
2-28	PTEHI and PTELO Registers—Extended Addressing	2-37
2-29	Instruction Cache Throttling Control Register (ICTC).....	2-38
2-30	Monitor Mode Control Register 0 (MMCR0).....	2-39

Figures

Figure Number	Title	Page Number
2-31	Monitor Mode Control Register 1 (MMCR1).....	2-42
2-32	Monitor Mode Control Register 2 (MMCR2).....	2-42
2-33	Breakpoint Address Mask Register (BAMR)	2-43
2-34	Performance Monitor Counter Registers (PMC1–PMC6).....	2-44
2-35	Sampled Instruction Address Registers (SIAR).....	2-45
3-1	Cache/Core Memory Subsystem Integration	3-5
3-2	L1 Data Cache Organization	3-10
3-3	L1 Instruction Cache Organization	3-11
3-4	Read Transaction—MSSCR0[EIDIS] = 0	3-19
3-5	RWITM and Flush Transactions—MSSCR0[EIDIS] = 0.....	3-19
3-6	Write Transaction—MSSCR0[EIDIS] = 0.....	3-20
3-7	Clean Transaction—MSSCR0[EIDIS] = 0	3-20
3-8	Kill Transaction—MSSCR0[EIDIS] = 0	3-21
3-9	Read Transaction—MSSCR0[EIDIS] = 1	3-21
3-10	RWITM, Write, and Flush Transactions—MSSCR0[EIDIS] = 1	3-22
3-11	Clean Transaction—MSSCR0[EIDIS] = 1	3-22
3-12	Kill Transaction—MSSCR0[EIDIS] = 1	3-23
3-13	Read Transaction Snoop Hit on the Reservation Address Register	3-23
3-14	Reskill Transaction Snoop Hit on the Reservation Address Register	3-24
3-15	Other Transaction Snoop Hit on the Reservation Address Register	3-24
3-16	PLRU Replacement Algorithm.....	3-39
3-17	L2 Cache Organization	3-45
3-18	Random Number Generator for L2 Replacement Selection	3-54
4-1	Machine Status Save/Restore Register 0 (SRR0)	4-8
4-2	Machine Status Save/Restore Register 1 (SRR1)	4-9
4-3	Machine State Register (MSR)	4-9
5-1	MMU Conceptual Block Diagram for a 32-Bit Physical Address (Not the e600 Core)	5-5
5-2	e600 Core IMMU Block Diagram, 36-Bit Physical Addressing.....	5-7
5-3	e600 Core DMMU Block Diagram, 36-Bit Physical Addressing.....	5-8
5-4	e600 Core DMMU Block Diagram with Extended Block Size and Additional BATs	5-9
5-5	Address Translation Types for 32-Bit Physical Addressing	5-11
5-6	Address Translation Types for 36-Bit Physical Addressing	5-12
5-7	General Flow in Selection of Which Address Translation to Use	5-15
5-8	General Flow of Page Translation.....	5-17
5-9	Format of Upper BAT Register (BATU)—Extended Block Size	5-24
5-10	Format of Lower BAT Register (BATL)—Extended Addressing	5-24
5-11	Block Physical Address Generation—Extended Addressing	5-27
5-12	Block Physical Address Generation—Extended Block Size for a 36-Bit Physical Address.....	5-29

Figures

Figure Number	Title	Page Number
5-13	Block Address Translation Flow—Extended Addressing	5-30
5-14	Block Address Translation Flow—Extended Block Size for a 36-Bit Physical Address.....	5-31
5-15	Generation of Extended 36-Bit Physical Address for Page Address Translation	5-33
5-16	Page Table Entry Format—Extended Addressing	5-34
5-17	Segment Register and DTLB Organization	5-39
5-18	tlbie Instruction Execution and MPX Bus Snooping Flow	5-41
5-19	tlbsync Instruction Execution and MPX Bus Snooping Flow.....	5-43
5-20	Page Address Translation Flow—TLB Hit—Extended Addressing	5-45
5-21	SDR1 Register Format—Extended Addressing.....	5-47
5-22	Hashing Functions for Page Table Entry Group Address	5-50
5-23	PTEG Address Generation for a Page Table Search—Extended Addressing.....	5-51
5-24	Example Page Table Structure—Extended Addressing	5-52
5-25	Example Primary PTEG Address Generation	5-54
5-26	Example Secondary PTEG Address Generation	5-55
5-27	Primary Page Table Search—Conceptual Flow	5-59
5-28	Secondary Page Table Search Flow—Conceptual Flow.....	5-60
5-29	Derivation of Key Bit for SRR1	5-63
5-30	TLBMISS Register	5-64
5-31	PTEHI and PTELO Registers—Extended Addressing	5-64
5-32	Flow for Example Software Table Search Operation	5-67
5-33	Flow for Generation of PTEG Address.....	5-68
5-34	Check and Set R and C Bit Flow	5-69
5-35	Page Fault Setup Flow	5-70
5-36	Setup for Protection Violation Interrupts	5-71
6-1	Pipelined Execution Unit	6-4
6-2	Superscalar/Pipeline Diagram.....	6-5
6-3	Stages and Events.....	6-8
6-4	e600 Core Pipeline Stages.....	6-9
6-5	BTIC Organization.....	6-11
6-6	Alignment of Target Instructions in the BTIC	6-12
6-7	Instruction Flow Diagram	6-14
6-8	Instruction Timing—Cache Hit.....	6-16
6-9	Instruction Timing—Cache Miss	6-19
6-10	Instruction Timing—Instruction Cache Miss/L2 Cache Hit	6-21
6-11	Branch Folding.....	6-24
6-12	Removal of Fall-Through Branch Instruction.....	6-25
6-13	Branch Completion (LR/CTR Write-Back)	6-26
6-14	Branch Instruction Timing	6-29
6-15	Vector Floating-Point Compare Bypass Non-Blocking	6-35
6-16	Vector Float Compare Bypass Blocking	6-36

Figures

Figure Number	Title	Page Number
6-17	LSU Block Diagram.....	6-69
7-1	Vector Registers (VRs).....	7-2
7-2	Vector Status and Control Register (VSCR).....	7-3
7-3	Vector Save/Restore Register (VRSERVE).....	7-4
8-1	Core Interface Signals.....	8-2
8-2	IEEE Std. 1149.1a-1993 Boundary-Scan Interface.....	8-6
8-3	e600 Core Block Diagram.....	8-8
9-1	e600 Core Power Management State Diagram.....	9-2
9-2	Example Core Power Management Handshaking.....	9-7
9-3	Instruction Cache Throttling Control Register (ICTC).....	9-10
10-1	Monitor Mode Control Register 0 (MMCR0).....	10-5
10-2	Monitor Mode Control Register 1 (MMCR1).....	10-8
10-3	Monitor Mode Control Register 2 (MMCR2).....	10-9
10-4	Breakpoint Address Mask Register (BAMR).....	10-9
10-5	Performance Monitor Counter Registers (PMC1–PMC6).....	10-10
10-6	Sampled Instruction Address Register (SIAR).....	10-11

Tables

Table Number	Title	Page Number
i	Acronyms and Abbreviated Terms.....	xxxvii
ii	Terminology Conventions.....	xli
iii	Instruction Field Conventions.....	xlii
1-1	e600 Core Interrupt Classifications.....	1-28
1-2	Interrupts and Exception Conditions.....	1-29
2-1	e600 Core Register Summary	2-3
2-2	PVR Settings	2-8
2-3	Additional PVR Bits	2-8
2-4	MSR Bit Settings	2-9
2-5	IEEE Std. 754 Floating-Point Exception Mode Bits.....	2-11
2-6	SDR1 Register Bit Settings—Extended Addressing	2-13
2-7	HID0 Field Descriptions	2-14
2-8	HID1 Field Descriptions	2-19
2-9	MSSCR0 Field Descriptions	2-21
2-10	MSSSR0 Field Descriptions	2-22
2-11	L2CR Field Descriptions	2-24
2-12	L2ERRINJHI Field Description.....	2-25
2-13	L2ERRINJLO Field Description	2-25
2-14	L2ERRINJCTL Field Descriptions.....	2-26
2-15	L2CAPTDATAHI Field Description.....	2-26
2-16	L2CAPTDATALO Field Description.....	2-27
2-17	L2CAPTECC Field Descriptions	2-27
2-18	L2ERRDET Field Descriptions	2-28
2-19	L2ERRDIS Field Descriptions.....	2-29
2-20	L2ERRINTEN Field Descriptions	2-30
2-21	L2ERRATTR Field Descriptions	2-31
2-22	L2ERRADDR Field Description	2-32
2-23	L2ERREADDR Field Description.....	2-32
2-24	L2ERRCTL Field Descriptions	2-33
2-25	ICTRL Field Descriptions.....	2-33
2-26	LDSTCR Field Descriptions.....	2-35
2-27	IABR Field Descriptions.....	2-35
2-28	TLBMISS Register—Field and Bit Descriptions	2-36
2-29	PTEHI and PTELO Bit Definitions.....	2-37
2-30	ICTC Field Descriptions	2-38
2-31	MMCR0 Field Descriptions.....	2-39
2-32	MMCR1 Field Descriptions.....	2-42
2-33	MMCR2 Field Descriptions.....	2-43

Tables

Table Number	Title	Page Number
2-34	BAMR Field Descriptions	2-43
2-35	PMC _n Field Descriptions	2-44
2-36	Settings Caused by Hard Reset (Used at Power-On)	2-46
2-37	Control Registers Synchronization Requirements	2-55
2-38	Integer Arithmetic Instructions	2-59
2-39	Integer Compare Instructions	2-60
2-40	Integer Logical Instructions	2-61
2-41	Integer Rotate Instructions	2-62
2-42	Integer Shift Instructions	2-62
2-43	Floating-Point Arithmetic Instructions	2-63
2-44	Floating-Point Multiply-Add Instructions	2-63
2-45	Floating-Point Rounding and Conversion Instructions	2-64
2-46	Floating-Point Compare Instructions	2-64
2-47	Floating-Point Status and Control Register Instructions	2-64
2-48	Floating-Point Move Instructions	2-65
2-49	Integer Load Instructions	2-67
2-50	Integer Store Instructions	2-68
2-51	Integer Load and Store with Byte-Reverse Instructions	2-69
2-52	Integer Load and Store Multiple Instructions	2-69
2-53	Integer Load and Store String Instructions	2-70
2-54	Floating-Point Load Instructions	2-70
2-55	Floating-Point Store Instructions	2-71
2-56	Store Floating-Point Single Behavior	2-71
2-57	Store Floating-Point Double Behavior	2-72
2-58	Branch Instructions	2-73
2-59	Condition Register Logical Instructions	2-74
2-60	Trap Instructions	2-74
2-61	System Linkage Instruction—UISA	2-74
2-62	Move To/From Condition Register Instructions	2-75
2-63	Move To/From Special-Purpose Register Instructions (UISA)	2-75
2-64	User-Level PowerPC SPR Encodings	2-76
2-65	User-Level SPR Encodings for e600-Defined Registers	2-76
2-66	Memory Synchronization Instructions—UISA	2-77
2-67	Move From Time Base Instruction	2-78
2-68	Memory Synchronization Instructions—VEA	2-79
2-69	User-Level Cache Instructions	2-80
2-70	System Linkage Instructions—OEA	2-82
2-71	Segment Register Manipulation Instructions (OEA)	2-82
2-72	Move To/From Machine State Register Instructions	2-82
2-73	Move To/From Special-Purpose Register Instructions (OEA)	2-83
2-74	Supervisor-Level PowerPC SPR Encodings	2-83

Tables

Table Number	Title	Page Number
2-75	Supervisor-Level SPR Encodings for e600-Defined Registers	2-85
2-76	Supervisor-Level Cache Management Instruction	2-87
2-77	Translation Lookaside Buffer Management Instruction	2-87
2-78	Vector Integer Arithmetic Instructions	2-92
2-79	CR6 Field Bit Settings for Vector Integer Compare Instructions	2-94
2-80	Vector Integer Compare Instructions	2-94
2-81	Vector Integer Logical Instructions	2-95
2-82	Vector Integer Rotate Instructions	2-95
2-83	Vector Integer Shift Instructions	2-95
2-84	Vector Floating-Point Arithmetic Instructions	2-96
2-85	Vector Floating-Point Multiply-Add Instructions	2-96
2-86	Vector Floating-Point Rounding and Conversion Instructions	2-97
2-87	Vector Floating-Point Compare Instructions	2-97
2-88	Vector Floating-Point Estimate Instructions	2-97
2-89	Vector Integer Load Instructions	2-98
2-90	Vector Load Instructions Supporting Alignment	2-99
2-91	Vector Integer Store Instructions	2-99
2-92	Vector Pack Instructions	2-100
2-93	Vector Unpack Instructions	2-100
2-94	Vector Merge Instructions	2-101
2-95	Vector Splat Instructions	2-101
2-96	Vector Permute Instruction	2-101
2-97	Vector Select Instruction	2-102
2-98	Vector Shift Instructions	2-102
2-99	Move To/From VSCR Register Instructions	2-102
2-100	AltiVec User-Level Cache Instructions	2-103
3-1	L1 Data Cache Status Bits	3-15
3-2	L2 Cache Status Bits	3-15
3-3	Snoop Response Summary	3-16
3-4	Snoop Intervention Summary	3-17
3-5	Simplified Transaction Types	3-17
3-6	Load and Store Ordering with WIMG Bit Settings	3-25
3-7	L1 PLRU Replacement Way Selection	3-39
3-8	PLRU Bit Update Rules	3-40
3-9	PLRU Bit Update Rules for AltiVec LRU Instructions	3-40
3-10	Definitions for L1 Cache-State Summary	3-42
3-11	L1 Cache-State Transitions and MSS Requests	3-43
3-12	L2 Cache Access Priorities	3-53
3-13	Definitions for L2 Cache-State Summary	3-55
3-14	L2 Cache State Transitions for Load, lwarx , Touch, and IFetches	3-56
3-15	L2 Cache State Transitions for Store Touch Operations	3-56

Tables

Table Number	Title	Page Number
3-16	L2 Cache State Transitions for Store (and stwx.) Operations	3-56
3-17	L2 Cache State Transitions for L1 Castout Operations.....	3-57
3-18	L2 Cache State Transitions for L2 Castout Operations.....	3-57
3-19	L2 Cache State Transitions for dcbf Operations.....	3-57
3-20	L2 Cache State Transitions for dcbz Operations	3-57
3-21	L2 Cache State Transitions for dcbst Operations	3-58
3-22	L2 Cache State Transitions for Write with Clean Operations.....	3-58
3-23	L2 Cache State Transitions for Remaining Instructions	3-58
3-24	MPX Bus Operations Caused by Cache Control Instructions (WIMG = xx1x)	3-59
3-25	MPX Bus Operations Caused by Cache Control Instructions (WIMG = xx0x)	3-60
3-26	Address/Transfer Attributes Generated by the e600 Core	3-61
3-27	Snooped Bus Transaction Summary	3-63
3-28	Definitions of Snoop Type for L1 Cache/Snoop Summary	3-64
3-29	Definitions of Other Terms for L1 Cache/Snoop Summary	3-64
3-30	L1 Cache State Transitions Due to Snoops	3-65
3-31	Definitions for L2 Cache/Snoop Summary.....	3-65
3-32	External Snoop Responses and L1 and L2 Actions	3-66
4-1	e600 Core Interrupt Classifications.....	4-3
4-2	Interrupts and Exception Conditions.....	4-3
4-3	e600 Interrupt Priorities	4-6
4-4	MSR Bit Settings	4-9
4-5	IEEE Std. 754 Floating-Point Exception Mode Bits.....	4-11
4-6	MSR Setting Due to Interrupt	4-14
4-7	System Reset Interrupt—Register Settings.....	4-16
4-8	Machine Check Enable Bits.....	4-17
4-9	Machine Check Interrupt—Register Settings	4-19
4-10	DSI Interrupt—Register Settings	4-21
4-11	External Interrupt—Register Settings	4-22
4-12	Alignment Interrupt—Register Settings	4-23
4-13	Performance Monitor Interrupt—Register Settings.....	4-26
4-14	TLB Miss Interrupts—Register Settings.....	4-27
4-15	Instruction Address Breakpoint Interrupt—Register Settings	4-28
4-16	System Management Interrupt—Register Settings.....	4-29
4-17	Altivec Assist Interrupt—Register Settings	4-30
5-1	MMU Features Summary.....	5-3
5-2	Access Protection Options for Pages	5-13
5-3	Translation Interrupt Conditions	5-18
5-4	Other MMU Exception Conditions.....	5-20
5-5	e600 Core Instruction Summary—Control MMUs	5-21
5-6	e600 Core MMU Registers	5-22
5-7	BAT Registers—Field and Bit Descriptions for Extended Addressing	5-25

Tables

Table Number	Title	Page Number
5-8	Upper BAT Register Block Size Mask Encoding	5-26
5-9	Upper BAT Register Block Size Mask Encoding when the Extended Block Size is Enabled (HID0[XBSEN] = 1)	5-28
5-10	PTE Bit Definitions	5-34
5-11	Table Search Operations to Update History Bits—TLB Hit Case	5-35
5-12	Model for Guaranteed R and C Bit Settings	5-37
5-13	SDR1 Register Bit Settings—Extended Addressing	5-47
5-14	Minimum Recommended Page Table Sizes—Extended Addressing	5-48
5-15	Implementation-Specific Resources for Software Table Search Operations	5-62
5-16	Implementation-Specific SRR1 Bits	5-63
5-17	TLBMISS Register—Field and Bit Descriptions	5-64
5-18	PTEHI and PTELO Bit Definitions	5-65
6-1	Performance Effects of Memory Operand Placement	6-32
6-2	Branch Operation Execution Latencies	6-37
6-3	System Operation Instruction Execution Latencies	6-37
6-4	Condition Register Logical Execution Latencies	6-38
6-5	Integer Unit Execution Latencies	6-39
6-6	Floating-Point Unit (FPU) Execution Latencies	6-41
6-7	Load/Store Unit (LSU) Instruction Latencies	6-42
6-8	Altivec Instruction Latencies	6-45
6-9	Fetch Alignment Example	6-51
6-10	Loop Example—Three Iterations	6-52
6-11	Branch-Taken Bubble Example	6-52
6-12	Eliminating the Branch-Taken Bubble	6-53
6-13	Misprediction Example	6-53
6-14	Three Iterations of Code Loop	6-54
6-15	Code Loop Example Using CTR	6-55
6-16	Link Stack Example	6-57
6-17	Position-Independent Code Example	6-57
6-18	Dispatch Stall Due to Rename Availability	6-59
6-19	Load/Store Multiple Micro-Operation Generation Example	6-60
6-20	GIQ Timing Example	6-62
6-21	VIQ Timing Example	6-62
6-22	Serialization Example	6-64
6-23	IU1 Timing Example	6-64
6-24	FPU Timing Example	6-66
6-25	FPSCR Rename Timing Example	6-67
6-26	Vector Execution Latencies	6-67
6-27	Vector Unit Example	6-68
6-28	Load Hit Pipeline Example	6-70
6-29	Store Hit Pipeline Example	6-70

Tables

Table Number	Title	Page Number
6-30	Execution of Four stfd Instructions	6-71
6-31	Load/Store Interaction (Assuming Full Alias)	6-71
6-32	Misaligned Load/Store Detection	6-72
6-33	Data Cache Miss, L2 Cache Hit Timing	6-72
6-34	Data Cache Miss, L2 Cache Miss, Main Memory Fetch Timing	6-73
6-35	Load Miss Line Alias Example	6-73
6-36	Load Miss Line Alias Example With Reordered Code	6-74
6-37	Store Miss Pipeline Example	6-75
6-38	Timing for Load Miss Line Alias Example	6-78
6-39	Hardware Prefetching Enable Example	6-78
7-1	VSCR Field Descriptions	7-3
7-2	VRSAVE Bit Settings	7-4
7-3	AltiVec User-Level Cache Instructions	7-6
7-4	Opcodes for dstx Instructions	7-7
7-5	DST[STRM] Description	7-7
7-6	The dstx Stream Termination Conditions	7-8
7-7	Denormalization for AltiVec Instructions	7-10
7-8	Vector Floating-Point Compare, Min, and Max in Non-Java Mode	7-11
7-9	Vector Floating-Point Compare, Min, and Max in Java Mode	7-12
7-10	Round-to-Integer Instructions in Non-Java Mode	7-12
7-11	Round-to-Integer Instructions in Java Mode	7-13
7-12	e600 Core AltiVec Features	7-14
7-13	Locations of e600 Core Instruction Execution	7-14
8-1	Summary of Selected Internal Signals	8-2
8-2	e600 Reset Configuration Signals	8-4
8-3	Transfer Type Encodings	8-11
9-1	Power Management Signals of the e600 Core	9-1
9-2	Power Management State Transitions	9-2
9-3	Core Power Management Control Bits	9-6
9-4	Required System Address Acknowledge Delay for Ratios < 5:1	9-9
9-5	ICTC Field Descriptions	9-10
10-1	Performance Monitor SPRs—Supervisor Level	10-4
10-2	Performance Monitor SPRs—User Level (Read-Only)	10-4
10-3	MMCR0 Field Descriptions	10-5
10-4	MMCR1 Field Descriptions	10-8
10-5	MMCR2 Field Descriptions	10-9
10-6	BAMR Field Descriptions	10-10
10-7	PMC _n Field Descriptions	10-10
10-8	Monitorable States	10-12
10-9	PMC1 Events—MMCR0[PMC1SEL] Select Encodings	10-13
10-10	PMC2 Events—MMCR0[PMC2SEL] Select Encodings	10-19

Tables

Table Number	Title	Page Number
10-11	PMC3 Events—MMCR1[PMC3SEL] Select Encodings	10-23
10-12	PMC4 Events—MMCR1[PMC4SEL] Select Encodings	10-25
10-13	PMC5 Events—MMCR1[PMC5SEL] Select Encodings	10-27
10-14	PMC6 Events—MMCR1[PMC6SEL] Select Encodings	10-28
A-1	Instructions by Mnemonic (Dec, Hex).....	A-1
A-2	Instructions by Primary and Secondary Opcodes (Dec, Hex)	A-12
A-3	Instructions by Mnemonic (Bin)	A-23
A-4	Instructions by Primary and Secondary Opcode (Bin)	A-34
A-5	Integer Arithmetic Instructions	A-45
A-6	Integer Compare Instructions	A-46
A-7	Integer Logical Instructions	A-46
A-8	Integer Rotate Instructions	A-47
A-9	Integer Shift Instruction	A-47
A-10	Floating-Point Arithmetic Instructions	A-48
A-11	Floating-Point Multiply-Add Instructions	A-48
A-12	Floating-Point Rounding and Conversion Instructions.....	A-49
A-13	Floating-Point Compare Instructions	A-49
A-14	Floating-Point Status and Control Register Instructions.....	A-49
A-15	Integer Load Instructions	A-50
A-16	Integer Store Instructions	A-50
A-17	Integer Load and Store with Byte Reverse Instructions.....	A-51
A-18	Integer Load and Store Multiple Instructions	A-51
A-19	Integer Load and Store String Instructions	A-51
A-20	Memory Synchronization Instructions.....	A-51
A-21	Floating-Point Load Instructions	A-52
A-22	Floating-Point Store Instructions	A-52
A-23	Floating-Point Move Instructions	A-52
A-24	Branch Instructions	A-52
A-25	Condition Register Logical Instructions	A-53
A-26	System Linkage Instructions	A-53
A-27	Trap Instructions	A-53
A-28	Processor Control Instructions	A-53
A-29	Cache Management Instructions.....	A-54
A-30	Segment Register Manipulation Instructions.....	A-54
A-31	Lookaside Buffer Management Instructions	A-54
A-32	Vector Integer Arithmetic Instructions.....	A-55
A-33	Floating-Point Compare Instructions	A-57
A-34	Floating-Point Estimate Instructions.....	A-57
A-35	Vector Load Instructions Supporting Alignment	A-57
A-36	Integer Store Instructions	A-57
A-37	Vector Pack Instructions.....	A-57

Tables

Table Number	Title	Page Number
A-38	Vector Unpack Instructions	A-58
A-39	Vector Splat Instructions	A-58
A-40	Vector Permute Instruction.....	A-58
A-41	Vector Select Instruction	A-59
A-42	Vector Shift Instructions.....	A-59
A-43	Move To/From Condition Register Instructions	A-59
A-44	User-Level Cache Instructions	A-59
A-45	I-Form	A-59
A-46	B-Form	A-60
A-47	SC-Form.....	A-60
A-48	D-Form.....	A-60
A-48	Table A-49. D-Form (continued)	A-61
A-49	X-Form.....	A-62
A-49	Table A-50. X-Form.....	A-63
A-50	XL-Form	A-67
A-51	XFX-Form.....	A-67
A-52	XFL-Form	A-68
A-53	XO-Form.....	A-68
A-54	A-Form.....	A-69
A-55	M-Form	A-70
A-56	VA-Form	A-70
A-57	VX-Form.....	A-71
A-58	VXR-Form	A-75
A-59	PowerPC Instruction Set Legend	A-75
B-1	32-Bit Instructions Not Implemented by the e600 Core	B-1
C-1	PowerPC SPR Encodings Ordered by Decimal Value.....	C-1
C-2	PowerPC SPR Encodings Ordered by Register Name.....	C-5

About This Book

The primary objective of this reference manual is to describe the functionality of the e600 core for software and hardware developers. The e600 core is a PowerPC™ core.

This book is intended as a companion to the *Programming Environments Manual for 32-Bit Implementations of the PowerPC Architecture* (referred to as the *Programming Environments Manual*), which provides a general description of the features that are common to processors and cores that implement the PowerPC architecture and indicates those features that are optional or that may be implemented differently in the design of each processor and core.

NOTE

About the Companion *Programming Environments Manual*

This manual, which describes e600 core features not defined by the architecture, is to be used with the *Programming Environments Manual*.

Because the PowerPC architecture definition is flexible to support a broad range of processors, the *Programming Environments Manual* describes generally those features common to these processors and indicates which features are optional or may be implemented differently in the design of each processor.

Note that the *Programming Environments Manual* describes only PowerPC architecture features for 32-bit implementations.

Go to www.freescale.com or contact a local sales representative for a copy of the *Programming Environments Manual*.

This manual and the *Programming Environments Manual* distinguish between the three levels, or programming environments, of the PowerPC architecture, which are as follows:

- PowerPC user instruction set architecture (UISA)—The UISA defines the architecture level to which user-level software should conform. The UISA defines the base user-level instruction set, user-level registers, data types, memory conventions, and the memory and programming models seen by application programmers.
- PowerPC virtual environment architecture (VEA)—The VEA, which is the smallest component of the PowerPC architecture, defines additional user-level functionality that falls outside typical user-level software requirements. The VEA describes the memory model for an environment in which multiple processors or other devices can access external memory and defines aspects of the cache model and cache control instructions from a user-level perspective. VEA resources are particularly useful for optimizing memory accesses and for managing resources in an environment in which other processors and other devices can access external memory.

Implementations that conform to the VEA also conform to the UISA but may not necessarily adhere to the OEA.



- PowerPC operating environment architecture (OEA)—The OEA defines supervisor-level resources typically required by an operating system. It defines the memory management model, supervisor-level registers, and the interrupt model.

Implementations that conform to the OEA also conform to the UISA and VEA.

Note that some resources are defined more generally at one level in the architecture and more specifically at another. For example, exception conditions that cause a program interrupt are defined by the UISA, but the interrupt mechanism itself is defined by the OEA.

Because it is important to distinguish between the levels of the architecture to ensure compatibility across multiple platforms, those distinctions are shown clearly throughout this book.

For ease in reference, topics in this book are presented in the same order as the *Programming Environments Manual*. Topics build on one another, beginning with a description and complete summary of the e600 core programming model (registers and instructions) and progressing to more specific, architecture-based topics regarding the cache, interrupt, and memory management models. As such, chapters may include information from multiple levels of the architecture. For example, the discussion of the cache model uses information from both the VEA and the OEA.

Additionally, the e600 core implements the AltiVec™ technology resources. The following two books describe the AltiVec technology:

- *AltiVec Technology Programming Environments Manual* (AltiVec PEM) is a reference guide for programmers. The AltiVec PEM uses a standardized format instruction to describe each instruction, showing syntax, instruction format, register translation language (RTL) code that describes how the instruction works, and a listing of which, if any, registers are affected. At the bottom of each instruction entry is a figure that shows the operations on elements within source operands and where the results of those operations are placed in the destination operand.
- *AltiVec Technology Programming Interface Manual* (AltiVec PIM) describes how programmers can access AltiVec functionality from programming languages such as C and C++. The AltiVec PIM describes the high-level language interface and application binary interface for System V and embedded applications for use with the AltiVec instruction set extension to the PowerPC architecture.

The *PowerPC Architecture: A Specification for a New Family of RISC Processors* defines the architecture from the perspective of the three programming environments and remains the defining document for the PowerPC architecture. For information on “ordering Freescale documentation, see [“Related Documentation,”](#) on page xxxvi.

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

To locate any published errata or updates for this document, refer to the world-wide web at <http://www.freescale.com>.

A list of the major differences between revisions of this manual is provided in [Appendix D, “Revision History.”](#)

Audience

This manual is intended to be used as a reference for many semiconductor products targeting a range of markets including automotive, communication, consumer, networking, and computer peripherals. It is intended for system software and hardware developers and applications programmers who want to develop products using the e600 core. It is assumed that the reader understands operating systems, core system design, basic principles of RISC processing, and details of the PowerPC architecture.

Organization

The following is a summary and a brief description of the chapters in this manual:

- [Chapter 1, “Overview,”](#) is useful for readers who want a general understanding of the features and functions of the PowerPC architecture and the e600 core. This chapter describes the flexible nature of the PowerPC architecture definition and provides an overview of how the PowerPC architecture defines the register set, operand conventions, addressing modes, instruction set, cache model, interrupt model, and memory management model.
- [Chapter 2, “Registers,”](#) is useful for software engineers who need to understand the e600-specific registers, operand conventions, and details regarding how PowerPC instructions are implemented on the e600 core. Instructions are organized by function.
- [Chapter 3, “L1 and L2 Cache Operation,”](#) discusses the cache and memory model as implemented on the e600 core.
- [Chapter 4, “Interrupts,”](#) describes the interrupt model defined in the OEA and the specific interrupt model implemented on the e600 core.
- [Chapter 5, “Memory Management,”](#) describes the implementation of the memory management unit on the e600 core as specified by the OEA.
- [Chapter 6, “Instruction Timing,”](#) provides information about latencies, interlocks, special situations, and various conditions that help make programming more efficient. This chapter is of special interest to software engineers and system designers.
- [Chapter 7, “AltiVec Technology Implementation,”](#) summarizes the features and functionality provided by the implementation of the AltiVec technology.
- [Chapter 8, “Core Interface,”](#) provides descriptions of individual signals of the e600 core. It also provides information on the core interface of the e600 core.
- [Chapter 9, “Power and Thermal Management,”](#) provides information about power saving and thermal management for the e600 core.
- [Chapter 10, “Performance Monitor,”](#) describes the operation of the performance monitor diagnostic tool incorporated in the e600 core.
- [Appendix A, “e600 Core Instruction Set Listings,”](#) lists all PowerPC instructions while indicating those instructions that are not implemented by the e600 core; it also includes the instructions that are specific to the e600 core. Instructions are grouped according to mnemonic, opcode, function, and form. A quick reference table is included that contains general information, such as the architecture level, privilege level, and form, and indicates if the instruction is 64-bit and optional.
- [Appendix B, “Instructions Not Implemented,”](#) provides a list of the 32- and 64-bit PowerPC instructions not implemented in the e600 core.



- [Appendix C, “Special-Purpose Registers,”](#) lists all e600 SPRs.
- [Appendix D, “Revision History,”](#) lists the major differences between revisions of the *e600 PowerPC™ Core Reference Manual*.
- This manual also includes a glossary and an index.

Suggested Reading

This section lists additional reading that provides background for the information in this manual as well as general information about the PowerPC architecture.

General Information

The following documentation, available through Morgan-Kaufmann Publishers, 340 Pine Street, Sixth Floor, San Francisco, CA, provides useful information about the PowerPC architecture and computer architecture in general:

- *The PowerPC Architecture: A Specification for a New Family of RISC Processors*, Second Edition, by International Business Machines, Inc.
For updates to the specification, see <http://www.austin.ibm.com/tech/ppc-chg.html>.
- *PowerPC Microprocessor Common Hardware Reference Platform: A System Architecture*, by Apple Computer, Inc., International Business Machines, Inc., and Motorola, Inc.
- *Computer Architecture: A Quantitative Approach*, Third Edition, by John L. Hennessy and David A. Patterson.
- *Computer Organization and Design: The Hardware/Software Interface*, Third Edition, David A. Patterson and John L. Hennessy.

Related Documentation

Freescale documentation is available from the sources listed on the back cover of this manual; the document order numbers are included in parentheses for ease in ordering:

- *Programming Environments Manual for 32-Bit Implementations of the PowerPC Architecture* (MPCFPE32B)—Describes resources defined by the PowerPC architecture.
- Reference manuals—These books provide details about individual implementations and are intended for use with the *Programming Environments Manual*.
- Addenda/errata to reference manuals—Because some processors have follow-on parts, an addendum is provided that describes additional features and functionality changes. These addenda are intended for use with the corresponding reference manuals.
- Hardware specifications—Hardware specifications provide specific data regarding bus timing, signal behavior, and AC, DC, and thermal characteristics, as well as other design considerations. Separate hardware specifications are provided for each part that includes one or more e600 cores. Note that when referring to the hardware specifications throughout this book, make sure to refer to the appropriate hardware specifications for the part being used.
- Application notes—These short documents address specific design issues useful to programmers and engineers working with Freescale processors.

Additional literature is published as new processors become available. For a current list of documentation, refer to <http://www.freescale.com>.

Conventions

This manual uses the following notational conventions:

cleared/set	When a bit takes the value zero, it is said to be cleared; when it takes a value of one, it is said to be set.	
mnemonics	Instruction mnemonics are shown in lowercase bold	
<i>italics</i>	Italics indicate variable command parameters, for example, bcctrx	
	Book titles in text are set in italics	
	Internal signals are set in italics, for example, <i>qual BG</i>	
0x0	Prefix to denote hexadecimal number	
0b0	Prefix to denote binary number	
rA, rB	Instruction syntax used to identify a source GPR	
rD	Instruction syntax used to identify a destination GPR	
frA, frB, frC	Instruction syntax used to identify a source FPR	
frD	Instruction syntax used to identify a destination FPR	
REG[FIELD]	Abbreviations for registers are shown in uppercase text. Specific bits, fields, or ranges appear in brackets. For example, MSR[LE] refers to the little-endian mode enable bit in the machine state register.	
x	In some contexts, such as signal encodings, an unitalicized x indicates a don't care.	
<i>x</i>	An italicized <i>x</i> indicates an alphanumeric variable	
<i>n</i>	An italicized <i>n</i> indicates an numeric variable	
¬	NOT logical operator	
&	AND logical operator	
	OR logical operator	
<table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>0000</td></tr></table>	0000	Indicates reserved bits or bit fields in a register. Although these bits can be written to as ones or zeros, they are always read as zeros.
0000		

Acronyms and Abbreviations

Table i contains acronyms and abbreviations that are used in this document.

Table i. Acronyms and Abbreviated Terms

Term	Meaning
ADB	Allowable disconnect boundary
ALU	Arithmetic logic unit

Table i. Acronyms and Abbreviated Terms (continued)

Term	Meaning
BAT	Block address translation
BHT	Branch history table
BIST	Built-in self test
BIU	Bus interface unit
BPU	Branch processing unit
BSDL	Boundary-scan description language
BTIC	Branch target instruction cache
CMOS	Complementary metal-oxide semiconductor
COP	Common on-chip processor
CQ	Completion queue
CR	Condition register
CTR	Count register
DABR	Data address breakpoint register
DAR	Data address register
DBAT	Data BAT
DCMP	Data TLB compare
DEC	Decrementer register
DLL	Delay-locked loop
DMISS	Data TLB miss address
DMMU	Data MMU
DPM	Dynamic power management
DSISR	Register used for determining the source of a DSI interrupt
DTLB	Data translation lookaside buffer
EA	Effective address
EAR	External access register
ECC	Error checking and correction
FIFO	First-in-first-out
FIQ	Floating-point issue queue
FPR	Floating-point register
FPSCR	Floating-point status and control register
FPU	Floating-point unit
GIQ	General-purpose register issue queue
GPR	General-purpose register

Table i. Acronyms and Abbreviated Terms (continued)

Term	Meaning
HID n	Hardware implementation-dependent register
IABR	Instruction address breakpoint register
IBAT	Instruction BAT
ICTC	Instruction cache throttling control register
IEEE	Institute for Electrical and Electronics Engineers
IMMU	Instruction MMU
IQ	Instruction queue
ITLB	Instruction translation lookaside buffer
IU	Integer unit
JTAG	Joint Test Action Group
L2	Secondary cache (level 2 cache)
L2CR	L2 cache control register
LIFO	Last-in-first-out
LR	Link register
LRU	Least recently used
LSB	Least-significant byte
lsb	Least-significant bit
LSQ	Least-significant quad word
lsq	Least-significant quad word
LSU	Load/store unit
MESI	Modified/exclusive/shared/invalid—cache coherency protocol
MMCR n	Monitor mode control registers
MMU	Memory management unit
MSB	Most-significant byte
msb	Most-significant bit
MSQ	Most-significant quad word
msq	Most-significant quad word
MSR	Machine state register
NaN	Not a number
No-op	No operation
OEA	Operating environment architecture
PEM	<i>Programming Environments Manual</i>
PID	Processor identification tag

Table i. Acronyms and Abbreviated Terms (continued)

Term	Meaning
PIM	<i>Programming Interface Manual</i>
PLL	Phase-locked loop
PLRU	Pseudo least recently used
PMC _n	Performance monitor counter registers
POR	Power-on reset
POWER	Performance optimized with enhanced RISC architecture
PTE	Page table entry
PTEG	Page table entry group
PVR	Processor version register
RAW	Read-after-write
RISC	Reduced instruction set computing
RTL	Register transfer language
RWITM	Read with intent to modify
RWNITM	Read with no intent to modify
SDA	Sampled data address register
SDR1	Register that specifies the page table base address for virtual-to-physical address translation
SIA	Sampled instruction address register
SPR	Special-purpose register
SR _n	Segment register
SRR0	Machine status save/restore register 0
SRR1	Machine status save/restore register 1
SRU	System register unit
TB	Time base facility
TBL	Time base lower register
TBU	Time base upper register
TLB	Translation lookaside buffer
TTL	Transistor-to-transistor logic
UIMM	Unsigned immediate value
UISA	User instruction set architecture
UMMCR _n	User monitor mode control registers
UPMC _n	User performance monitor counter registers
USIA	User sampled instruction address register
VEA	Virtual environment architecture

Table i. Acronyms and Abbreviated Terms (continued)

Term	Meaning
VFPU	Vector floating-point unit
VIQ	Vector issue queue
VIU1	Vector instruction unit 1
VIU2	Vector instruction unit 2
VPN	Virtual page number
VPU	Vector permute unit
VSID	Virtual segment identification
VTQ	Vector touch queue
WAR	Write-after-read
WAW	Write-after-write
WIMG	Write-through/caching-inhibited/memory-coherency enforced/guarded bits
XATC	Extended address transfer code
XER	Register used for indicating conditions such as carries and overflows for integer operations

Terminology Conventions

Table ii describes terminology conventions used in this manual and the equivalent terminology used in the PowerPC architecture specification.

Table ii. Terminology Conventions

The Architecture Specification	This Manual
Change bit	Changed bit
Data storage interrupt (DSI)	DSI interrupt
Extended mnemonics	Simplified mnemonics
Fixed-point unit (FXU)	Integer unit (IU)
Instruction storage interrupt (ISI)	ISI interrupt
Privileged mode (or privileged state)	Supervisor-level privilege
Problem mode (or problem state)	User-level privilege
Real address	Physical address
Reference bit	Referenced bit
Relocation	Translation
Storage (locations)	Memory
Storage (the act of)	Access
Store in	Write back
Store through	Write through

Table iii describes instruction field notation used in this manual.

Table iii. Instruction Field Conventions

The Architecture Specification	Equivalent to:
BA, BB, BT	crbA, crbB, crbD (respectively)
BF, BFA	crfD, crfS (respectively)
D	d
DS	ds
FLM	FM
FRA, FRB, FRC, FRT, FRS	frA, frB, frC, frD, frS (respectively)
FXM	CRM
RA, RB, RT, RS	rA, rB, rD, rS (respectively)
SI	SIMM
U	IMM
UI	UIMM
<i>/, //, ///</i>	0...0 (shaded)

Chapter 1

Overview

This chapter provides an overview of the e600 core features, including a block diagram showing the major functional components. The e600 is a PowerPC™ core.

This document also provides information about how the e600 implementation complies with the PowerPC and AltiVec™ architecture definitions.

1.1 e600 Core Overview

This section describes the features and general operation of the e600 core and provides a block diagram showing the major functional units. The e600 implements the PowerPC architecture and is a reduced instruction set computer (RISC) core. The e600 core includes separate 32-Kbyte L1 instruction and data caches and a 1-Mbyte L2 cache. The core is a high-performance superscalar design supporting multiple execution units, including four independent units that execute AltiVec instructions.

The e600 core implements the 32-bit portion of the PowerPC architecture, which provides 32-bit effective addresses, integer data types of 8, 16, and 32 bits, and floating-point data types of 32 (single precision) and 64 (double precision) bits. The core supports up to 4 Petabytes (2^{52}) of virtual memory and up to 64 Gigabytes (2^{36}) of physical memory.

The e600 core also implements the AltiVec instruction set architectural extension. The e600 core can dispatch and complete three instructions simultaneously. It incorporates the following execution units:

- 64-bit floating-point unit (FPU)
- Branch processing unit (BPU)
- Load/store unit (LSU)
- Four integer units (IUs):
 - Three shorter latency IUs (IU1a–IU1c)—execute all integer instructions except multiply, divide, and move to/from special-purpose register (SPR) instructions.
 - Longer latency IU (IU2)—executes miscellaneous instructions including condition register (CR) logical operations, integer multiplication and division instructions, and move to/from SPR instructions.
- Four vector units that support AltiVec instructions:
 - Vector permute unit (VPU)
 - Vector integer unit 1 (VIU1)—performs shorter latency integer calculations
 - Vector integer unit 2 (VIU2)—performs longer latency integer calculations
 - Vector floating-point unit (VFPU)

The ability to execute several instructions in parallel and the use of simple instructions with rapid execution times yield high efficiency and throughput for e600-based systems. Most integer instructions (including VIU1 instructions) have a one-clock cycle execution latency.

Several execution units feature multiple-stage pipelines; that is, the tasks they perform are broken into subtasks executed in successive stages. Typically, instructions follow one another through the stages, so a

four-stage unit can work on four instructions when its pipeline is full. So, although an instruction may have to pass through several stages, the execution unit can achieve a throughput of one instruction per clock cycle.

AltiVec computational instructions are executed in four independent, pipelined AltiVec execution units. A maximum of two AltiVec instructions can be issued out-of-order to any combination of AltiVec execution units per clock cycle from the bottom two VIQ entries (VIQ1–VIQ0). This means an instruction in VIQ1 does not have to wait for an instruction in VIQ0 that is waiting for operand availability. Moreover, the VIU2, VFPU, and VPU are pipelined, so they can operate on multiple instructions. Because VPU has a two-stage pipeline; the VIU2 and VFPU each have four-stage pipelines, as many as ten AltiVec instructions can be executing concurrently.

Note that for the e600 core, double- and single-precision versions of floating-point instructions have the same latency. For example, a floating-point multiply-add instruction takes 5 cycles to execute, regardless of whether it is single (**fmadds**) or double precision (**fmadd**).

The e600 core has independent 32-Kbyte, eight-way set-associative, physically addressed L1 (level one) caches for instructions and data, and independent instruction and data memory management units (MMUs). Each MMU has a 128-entry, two-way set-associative translation lookaside buffer (DTLB and ITLB) that saves recently used page address translations. Block address translation is implemented with the eight-entry instruction and data block address translation (IBAT and DBAT) arrays defined by the PowerPC architecture. During block translation, effective addresses are compared simultaneously with all BAT entries, as described in [Chapter 5, “Memory Management.”](#) For information about the L1 caches, see [Chapter 3, “L1 and L2 Cache Operation.”](#)

The L2 cache is implemented with 1-Mbyte, eight-way set-associative physically addressed memory within the core available for storing data, instructions, or both. The L2 cache supports parity generation and checking for both tags and data. If error checking and correction (ECC) is disabled, it responds with an 11.5-cycle load latency for an L1 miss that hits in the L2; if ECC is enabled, the L2 load access time is 12.5 cycles. The L2 cache is fully pipelined for two-cycle throughput. For information about the L2 cache implementation, see [Chapter 3, “L1 and L2 Cache Operation.”](#)

The three power-saving modes, nap, sleep, and deep sleep, progressively reduce power dissipation. When functional units are idle, a dynamic power management mode causes those units to enter a low-power mode automatically without affecting operational performance, software execution, or hardware external to the core. [Section 1.2.7, “Power and Thermal Management,”](#) describes how power management can be used to reduce power consumption when the core, or portions of it, are idle. It also describes how the instruction cache throttling mechanism reduces the instruction dispatch rate. Power management states are described more fully in [Chapter 9, “Power and Thermal Management.”](#)

The performance monitor facility provides the ability to monitor and count predefined events such as core clocks, misses in the instruction cache, data cache, or L2 cache, types of instructions dispatched, mispredicted branches, and other occurrences. The count of such events (which may be an approximation) can be used to trigger the performance monitor interrupt. [Section 1.2.8, “Core Performance Monitor,”](#) describes the operation of the performance monitor diagnostic tool. This functionality is fully described in [Chapter 10, “Performance Monitor.”](#)

[Figure 1-1](#) shows the parallel organization of the execution units (shaded in the diagram). Note that this is a conceptual model showing basic features, rather than an attempt at showing how features are implemented physically.

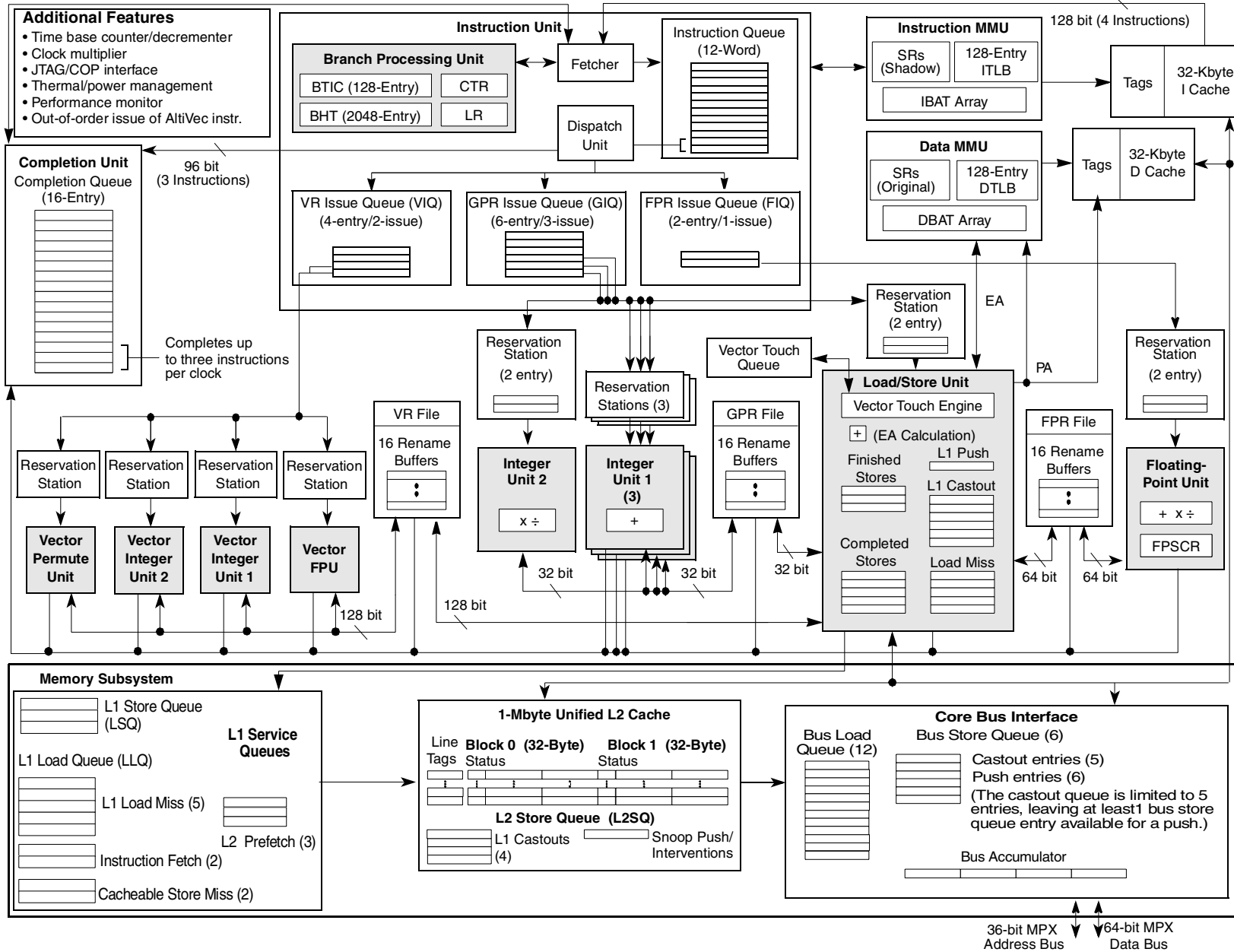


Figure 1-1. e600 Core Block Diagram

1.2 e600 Core Features

This section describes the features of the e600 core. The interrelationships of these features are shown in [Figure 1-1](#).

Major features of the e600 core are as follows:

- High-performance superscalar e600 core
 - As many as 4 instructions can be fetched from the instruction cache at a time.
 - As many as 3 instructions can be dispatched to the issue queues at a time.
 - As many as 12 instructions can be in the instruction queue (IQ).
 - As many as 16 instructions can be at some stage of execution simultaneously.
 - Single-cycle execution for most instructions
 - One-instruction throughput per clock cycle for most instructions
 - Seven-stage pipeline control
- Eleven independent execution units and three register files
 - Branch processing unit (BPU) features static and dynamic branch prediction
 - 128-entry (32-set, four-way set-associative) branch target instruction cache (BTIC), a cache of branch instructions that have been encountered in branch/loop code sequences. If a target instruction is in the BTIC, it is fetched into the instruction queue a cycle sooner than it can be made available from the instruction cache. Typically, a fetch that hits the BTIC provides the first 4 instructions in the target stream.
 - 2048-entry branch history table (BHT) with 2 bits per entry for four levels of prediction—strongly not-taken, not-taken, taken, strongly taken
 - Up to three outstanding speculative branches
 - Branch instructions that do not update the count register (CTR) or link register (LR) are often removed from the instruction stream.
 - Eight-entry link register stack to predict the target address of Branch Conditional to Link Register (**bclr**) instructions
 - Four integer units (IUs) that share 32 GPRs for integer operands
 - Three identical IUs (IU1a, IU1b, and IU1c) can execute all integer instructions except multiply, divide, and move to/from SPR instructions.
 - IU2 executes miscellaneous instructions including the CR logical operations, integer multiplication and division instructions, and move to/from special-purpose register instructions.
 - 64-bit floating-point unit (FPU)
 - Five-stage FPU
 - Designed to comply with IEEE Std. 754™-1985 FPU for both single- and double-precision operations
 - Supports non-IEEE Std. 754 mode for time-critical operations
 - Hardware support for denormalized numbers
 - Thirty-two 64-bit FPRs for single- or double-precision operands

- Four vector units and 32-entry vector register file (VRs)
 - Vector permute unit (VPU)
 - Vector integer unit 1 (VIU1) handles short-latency AltiVec integer instructions, such as vector add instructions (for example, **vaddsbs**, **vaddshs**, and **vaddsws**)
 - Vector integer unit 2 (VIU2) handles longer-latency AltiVec integer instructions, such as vector multiply add instructions (for example, **vmhaddshs**, **vmhraddshs**, and **vmladduhm**).
 - Vector floating-point unit (VFPU)
- Three-stage load/store unit (LSU)
 - Supports integer, floating-point and vector instruction load/store traffic
 - Four-entry vector touch queue (VTQ) supports all four architected AltiVec data stream operations
 - Three-cycle GPR and AltiVec load latency (byte, half word, word, vector) with single-cycle throughput
 - Four-cycle FPR load latency (single, double) with single-cycle throughput
 - No additional delay for misaligned access within double-word boundary
 - Dedicated adder calculates effective addresses (EAs)
 - Supports store gathering
 - Performs alignment, normalization, and precision conversion for floating-point data
 - Executes cache control and TLB instructions
 - Performs alignment, zero padding, and sign extension for integer data
 - Supports hits under misses (multiple outstanding misses)
 - Supports both big- and little-endian modes, including misaligned little-endian accesses
- Dispatch unit—The decode/dispatch stage fully decodes each instruction.
- The FIQ (floating-point issue queue), VIQ (vector issue queue), and GIQ (general purpose issue queue) can accept as many as one, two, and three instructions, respectively, in a cycle. Instruction dispatch requires the following:
 - Instructions can be dispatched only from the three lowest IQ entries—IQ0, IQ1, and IQ2.
 - A maximum of three instructions can be dispatched to the issue queues per clock cycle.
 - Space must be available in the completion queue (CQ) for an instruction to dispatch (this includes instructions that are assigned a space in the CQ but not in an issue queue).
- Rename buffers
 - 16 GPR (general purpose register) rename buffers
 - 16 FPR (floating-point register) rename buffers
 - 16 VR (vector register) rename buffers
- Completion unit
 - Retires an instruction from the 16-entry CQ when all instructions ahead of it have been completed, the instruction has finished execution, and no interrupts are pending.
 - Guarantees sequential programming model (precise interrupt model)

- Monitors all dispatched instructions and retires them in order
- Tracks unresolved branches and flushes instructions after a mispredicted branch
- Retires as many as three instructions per clock cycle
- L1 cache has the following characteristics:
 - Two separate 32-Kbyte instruction and data caches (Harvard architecture)
 - Instruction and data caches are eight-way set-associative
 - Instruction and data caches have 32-byte cache blocks. A cache block is the block of memory that a coherency state describes—it corresponds to a cache line for the L1 data cache.
 - Cache directories are physically addressed. The physical (real) address tag is stored in the cache directory.
 - The caches implement a pseudo least-recently-used (PLRU) replacement algorithm within each way.
 - Cache write-back or write-through operation is programmable on a per-page or per-block basis.
 - Instruction cache can provide four instructions per clock cycle; data cache can provide four words per clock cycle
 - Two-cycle latency and single-cycle throughput for instruction or data cache accesses
 - Caches can be disabled in software
 - Caches can be locked in software
 - Supports a four-state modified/exclusive/shared/invalid (MESI) coherency protocol
 - A single coherency status bit for each instruction cache block allows encoding for the following two possible states:
 - Invalid (INV)
 - Valid (VAL)
 - Three status bits for each data cache block allow encoding for coherency, as follows:
 - 0xx = invalid (I)
 - 101 = shared (S)
 - 100 = exclusive (E)
 - 110 = modified (M)
 - Separate copy of data cache tags for efficient snooping
 - Both L1 caches support parity generation and checking (enabled through bits in the instruction cache and interrupt control (ICTRL) register) as follows:
 - Instruction cache—one parity bit per instruction
 - Data cache—one parity bit per byte of data
 - No snooping of instruction cache except for **icbi** instruction
 - Caches implement a pseudo least-recently-used (PLRU) replacement algorithm within each way
 - Data cache supports AltiVec LRU and transient instructions, as described in [Section 1.3.2.2, “AltiVec Instruction Set”](#)

- Critical double- and/or quad-word forwarding is performed as needed. Critical quad-word forwarding is used for AltiVec loads and instruction fetches. Other accesses use critical double-word forwarding.
- Level 2 (L2) cache within the core has the following features:
 - Integrated 1-Mbyte, eight-way set-associative unified instruction and data cache
 - Pipelined to provide 32 bytes every other clock cycle to the L1 caches
 - Total latency of 11.5 processor cycles for L1 data cache miss that hits in the L2 with ECC disabled, 12.5 cycles when ECC is enabled
 - Uses one of two random replacement algorithms (selectable through L2CR)
 - Cache write-back or write-through operation programmable on a per-page or per-block basis
 - Organized as 32 bytes/block and 2 blocks (sectors)/line (a cache block is the block of memory that a coherency state describes).
 - Supports error correction and detection using a SECDED (single-error correction, double-error detection) protocol. Every 64 bits of data comes with 8 bits of error detection/correction, which can be programmed as ECC across the 64 bits of data, byte parity, or no error detection/correction.
 - Supports parity generation and checking for both tags and data (enabled through L2CR). Tag parity is enabled separately in the L2ERRDIS register, and data parity can be enabled through L2CR only when ECC is disabled.
 - Error injection modes provided for testing
- Separate memory management units (MMUs) for instructions and data
 - 52-bit virtual address; 32- or 36-bit physical address
 - Address translation for 4-Kbyte pages, variable-sized blocks, and 256-Mbyte segments
 - Memory programmable as write-back/write-through, caching-inhibited/caching-allowed, and memory coherency enforced/memory coherency not enforced on a page or block basis
 - Separate IBATs and DBATs (eight each) also defined as SPRs
 - Separate instruction and data translation lookaside buffers (TLBs)
 - Both TLBs are 128-entry, two-way set-associative, and use LRU replacement algorithm
 - TLBs are hardware or software reloadable (that is, on a TLB miss a page table search is performed in hardware or by system software).
- Efficient data flow
 - Although the VR/LSU interface is 128 bits, the L1/L2 bus interface allows up to 256 bits.
 - The L1 data cache is pipelined to provide 128 bits/cycle to or from the VRs.
 - L2 cache is fully pipelined to provide 32 bytes every other processor clock cycle to the L1 cache
 - As many as nine outstanding, out-of-order cache misses are allowed between the L1 data cache and L2 bus: up to 5 load or touch instructions, 2 cacheable store instructions, and/or 2 instruction fetches.
 - As many as 16 out-of-order transactions can be present on the MPX bus.

- Store merging for multiple store misses to the same line. Only coherency action taken (address-only) for store misses merged to all 32 bytes of a cache block (no data tenure needed)
- Support for a second cacheable store miss
- Three-entry finished store queue and five-entry completed store queue between the LSU and the L1 data cache
- Separate additional queues for efficient buffering of outbound data (such as castouts and write-through stores) from the L1 data cache and L2 cache
- Multiprocessing support features include the following:
 - Hardware-enforced, MESI cache coherency protocols for data cache
 - Load/store with reservation instruction pair for atomic memory references, semaphores, and other multiprocessor operations
- Power and thermal management
 - The following three power-saving modes are available to the system:
 - Nap—Instruction fetching is halted. Only those clocks for the time base, decremter, and JTAG logic remain running. The core goes into the *doze* state to snoop memory operations on the MPX bus and then back to nap using a *qreq/qack* processor-system handshake protocol.
 - Sleep—Power consumption is further reduced by disabling bus snooping, leaving only the PLL in a locked and running state. All internal functional units are disabled.
 - Deep sleep—When the core is in deep sleep state, the system can disable the PLL. The system can then disable the SYSCLK source for greater system power savings. Power-on reset procedures for restarting and relocking the PLL must be followed upon exiting the deep sleep state.
 - Dynamic frequency switching (DFS) capability conserves power by lowering core operating frequency. Divide-by-two and divide-by-four modes (DFS2 and DFS4) provided.
 - Instruction cache throttling provides control of instruction fetching to lower device temperature.
- Performance monitor helps to debug system designs and improve software efficiency
- In-system testability and debugging features through JTAG boundary-scan capability
- Reliability and serviceability
 - Parity checking on the MPX bus
 - Parity checking on L1 and L2 cache arrays

1.2.1 Instruction Flow

As shown in [Figure 1-1](#), the e600 core instruction unit provides centralized control of instruction flow to the execution units. The instruction unit contains a sequential fetcher, 12-entry instruction queue (IQ), dispatch unit, and branch processing unit (BPU). It determines the address of the next instruction to be fetched based on information from the sequential fetcher and from the BPU.

See [Chapter 6, “Instruction Timing,”](#) for a detailed discussion of instruction timing.

The sequential fetcher loads instructions from the instruction cache into the instruction queue. The BPU extracts branch instructions from the sequential fetcher. Branch instructions that cannot be resolved immediately are predicted using either e600-specific dynamic branch prediction or architecture-defined static branch prediction.

Branch instructions that do not affect the LR or CTR are often removed from the instruction stream. [Section 6.4.1.1, “Branch Folding and Removal of Fall-Through Branch Instructions,”](#) describes when a branch can be removed from the instruction stream.

Instructions dispatched beyond a predicted branch do not complete execution until the branch is resolved, preserving the programming model of sequential execution. If branch prediction is incorrect, the instruction unit flushes all predicted path instructions, and instructions are fetched from the correct path.

1.2.1.1 Instruction Queue and Dispatch Unit

The instruction queue (IQ), shown in [Figure 1-1](#) holds as many as 12 instructions and loads as many as 4 instructions from the instruction cache during a single processor clock cycle.

The fetcher attempts to initiate a new fetch every cycle. The two fetch stages are pipelined, so as many as four instructions can arrive to the IQ every cycle. All instructions except branch (**bx**), Return from Interrupt (**rfi**), System Call (**sc**), Instruction Synchronize (**isync**), and no-op instructions are dispatched to their respective issue queues from the dispatch entries in the instruction queue (IQ0–IQ2) at a maximum rate of three instructions per clock cycle. Reservation stations are provided for the three IU1s, IU2, FPU, LSU, VPU, VIU2, VIU1, and VFPU. The dispatch unit checks for source and destination register dependencies, determines whether a position is available in the CQ, and inhibits subsequent instruction dispatching as required.

Branch instruction can be detected, decoded, and predicted from entries IQ0–IQ7. See [Section 6.3.3, “Dispatch, Issue, and Completion Considerations.”](#)

1.2.1.2 Branch Processing Unit (BPU)

The BPU receives branch instructions from the IQ and executes them early in the pipeline, achieving the effect of a zero-cycle branch in some cases.

Branches with no outstanding dependencies (CR, LR, or CTR unresolved) can be processed and resolved immediately. For branches in which only the direction is unresolved due to a CR or CTR dependency, the branch path is predicted using either architecture-defined static branch prediction or e600-specific dynamic branch prediction. Dynamic branch prediction is enabled if HID0[BHT] is set. For **bclr** branches

where the target address is unresolved due to a LR dependency, the branch target can be predicted using the hardware link stack. Link stack prediction is enabled if `HID0[LRSTK]` is set.

When a prediction is made, instruction fetching, dispatching, and execution continue from the predicted path, but instructions cannot complete and write back results to architected registers until the prediction is determined to be correct (resolved). When a prediction is incorrect, the instructions from the incorrect path are flushed from the core and processing begins from the correct path.

Dynamic prediction is implemented using a 2048-entry branch history table (BHT), a cache that provides two bits per entry that together indicate four levels of prediction for a branch instruction—not-taken, strongly not-taken, taken, strongly taken. When dynamic branch prediction is disabled, the BPU uses a bit in the instruction encoding to predict the direction of the conditional branch. Therefore, when an unresolved conditional branch instruction is encountered, the e600 core executes instructions from the predicted target stream although the results are not committed to architected registers until the conditional branch is resolved. Unresolved branches are held in a three-entry branch queue. When the branch queue is full, no further conditional branches can be processed until one of the conditions in the branch queue is resolved.

When a branch is taken or predicted as taken, instructions from the untaken path must be flushed and the target instruction stream must be fetched into the IQ. The BTIC is a 128-entry, four-way set associative cache that contains the most recently used branch target instructions (up to four instructions per entry) for **b** and **bc** branches. When a taken branch instruction of this type hits in the BTIC, the instructions arrive in the IQ two clock cycles later, a clock cycle sooner than they would arrive from the instruction cache. Additional instructions arrive from the instruction cache in the next clock cycle. The BTIC reduces the number of missed opportunities to dispatch instructions and gives the core a 1-cycle head start on processing the target stream.

The BPU contains an adder to compute branch target addresses and three user-accessible registers—the link register (LR), the count register (CTR), and the condition register (CR). The BPU calculates the return pointer for subroutine calls and saves it in the LR for certain types of branch instructions. The LR also contains the branch target address for Branch Conditional to Link Register (**bclr_x**) instructions. The CTR contains the branch target address for Branch Conditional to Count Register (**bcctr_x**) instructions. Because the LR and CTR are SPRs, their contents can be copied to or from any GPR. Also, because the BPU uses dedicated registers rather than GPRs or FPRs, execution of branch instructions is largely independent from execution of integer and floating-point instructions.

1.2.1.3 Completion Unit

The completion unit operates closely with the instruction unit. Instructions are fetched and dispatched in program order. At the point of dispatch, the program order is maintained by assigning each dispatched instruction a successive entry in the 16-entry CQ. The completion unit tracks instructions from dispatch through execution and retires them in program order from the three bottom CQ entries (CQ0–CQ2).

Instructions cannot be dispatched to an execution unit unless there is a CQ vacancy.

Branch instructions that do not update the CTR or LR are often removed from the instruction stream. Those that are removed do not take a CQ entry. Branches that are not removed from the instruction stream

follow the same dispatch and completion procedures as non-branch instructions but are not dispatched to an issue queue.

Completing an instruction commits execution results to architected registers (GPRs, FPRs, VRs, LR, and CTR). In-order completion ensures the correct architectural state when the e600 core must recover from a mispredicted branch or any interrupt. An instruction is retired as it is removed from the CQ.

For a more detailed discussion of instruction completion, see [Section 6.3.3, “Dispatch, Issue, and Completion Considerations.”](#)

1.2.1.4 Independent Execution Units

In addition to the BPU, the e600 core provides the ten execution units described in the following sections.

1.2.1.4.1 Altivec Vector Permute Unit (VPU)

The VPU executes permutation instructions such as pack, unpack, merge, splat, and permute on vector operands.

1.2.1.4.2 Altivec Vector Integer Unit 1 (VIU1)

The VIU1 executes simple vector integer computational instructions, such as addition, subtraction, maximum and minimum comparisons, averaging, rotation, shifting, comparisons, and Boolean operations.

1.2.1.4.3 Altivec Vector Integer Unit 2 (VIU2)

The VIU2 executes longer-latency vector integer instructions, such as multiplication, multiplication/addition, and sum-across with saturation.

1.2.1.4.4 Altivec Vector Floating-Point Unit (VFPU)

The VFPU executes all vector floating-point instructions.

A maximum of two Altivec instructions can be issued out-of-order to any combination of Altivec execution units per clock cycle from the bottom two VIQ entries (VIQ1–VIQ0). An instruction in VIQ1 does not have to wait for an instruction in VIQ0 that is waiting for operand availability. Moreover, the VIU2, VFPU, and VPU are pipelined, so they can operate on multiple instructions.

1.2.1.4.5 Integer Units (IUs)

The integer units (three IU1s and IU2) are shown in [Figure 1-1](#). The IU1s execute shorter latency integer instructions, that is, all integer instructions except multiply, divide, and move to/from special-purpose register instructions. IU2 executes integer instructions with latencies of three cycles or more.

IU2 has a 32-bit integer multiplier/divider and a unit for executing CR logical operations and move to/from SPR instructions. The multiplier supports early exit for operations that do not require full 32×32 -bit multiplication.

1.2.1.4.6 Floating-Point Unit (FPU)

The FPU, shown in [Figure 1-1](#), is designed such that double-precision operations require only a single pass, with a latency of 5 cycles. As instructions are dispatched to the FPU's reservation station, source operand data can be accessed from the FPRs or from the FPR rename buffers. Results in turn are written to the rename buffers and made available to subsequent instructions. Instructions start execution from the bottom reservation station only and execute in program order.

The FPU contains a single-precision multiply-add array and the floating-point status and control register (FPSCR). The multiply-add array allows the e600 core to implement multiply and multiply-add operations efficiently. The FPU is pipelined so that one single- or double-precision instruction can be issued per clock cycle.

Note that an execution bubble occurs after four consecutive, independent floating-point arithmetic instructions execute to allow for a normalization special case. Thirty-two 64-bit floating-point registers are provided to support floating-point operations. Stalls due to contention for FPRs are minimized by automatic allocation of the 16 floating-point rename registers. The e600 core writes the contents of the rename registers to the appropriate FPR when floating-point instructions are retired by the completion unit.

The e600 core supports all IEEE Std. 754 floating-point data types (normalized, denormalized, NaN, zero, and infinity) in hardware, eliminating the latency incurred by software interrupt routines.

1.2.1.4.7 Load/Store Unit (LSU)

The LSU executes all load and store instructions as well as the AltiVec LRU and transient instructions and provides the data transfer interface between the GPRs, FPRs, VRs, and the cache/core memory subsystem (MSS). The LSU also calculates effective addresses and aligns data.

Load and store instructions are issued and translated in program order; however, some memory accesses can occur out of order. Synchronizing instructions can be used to enforce strict ordering. When there are no data dependencies and the guarded bit for the page or block is cleared, a maximum of one out-of-order cacheable load operation can execute per clock cycle from the perspective of the LSU. Loads to FPRs require a 4-cycle total latency. Data returned from the cache is held in a rename register until the completion logic commits the value to a GPR, FPR, or VR. Stores cannot be executed out of order and are held in the store queue until the completion logic signals that the store operation is to be completed to memory. The e600 core executes store instructions with a maximum throughput of one every three clock cycles and a 3-cycle total latency to the data cache. The time required to perform the load or store operation depends on the processor: bus clock ratio and whether the operation involves the caches, system memory, or an I/O device.

1.2.2 Memory Management Units (MMUs)

The e600 core MMUs support up to 4 Petabytes (2^{52}) of virtual memory and 64 Gigabytes (2^{36}) of physical memory for instructions and data. The MMUs control access privileges for these spaces on block and page granularities. Referenced and changed status is maintained by the core for each page to support demand-paged virtual memory systems. The MMUs are contained within the LSU.

The MMU translates the effective address calculated by the IU and LSU to determine the physical address for the memory access.

The e600 core supports the following types of memory translation:

- Real addressing mode—In this mode, translation is disabled by clearing bits in the machine state register (MSR): MSR[IR] for instruction fetching or MSR[DR] for data accesses. When address translation is disabled, the physical address is identical to the effective address. When extended addressing is disabled ($HID0[XAEN] = 0$) a 32-bit physical address is used, PA[4–35]. For more details, see [Section 5.1.3, “Address Translation Mechanisms.”](#)
- Page address translation—translates the page frame address for a 4-Kbyte page size
- Block address translation—translates the base address for blocks (4 Gbytes)

If translation is enabled, the appropriate MMU translates the higher-order bits of the effective address into physical address bits. Lower-order address bits are untranslated and are the same for both logical and physical addresses. These bits are directed to the caches where they form the index into the eight-way set-associative tag array. After translating the address, the MMU passes the higher-order physical address bits to the cache and the cache lookup completes. For caching-inhibited accesses or accesses that miss in the cache, the untranslated lower-order address bits are concatenated with the translated higher-order address bits; the resulting 32- or 36-bit physical address is used by the core memory subsystem (MSS) and the bus interface unit (BIU) to access memory external to the core.

The TLBs store page address translations for recent memory accesses. For each access, an effective address is presented for page and block translation simultaneously. If a translation is found in both the TLB and the BAT array, the block address translation in the BAT array is used. Usually the translation is in a TLB and the physical address is readily available to the cache. When a page address translation is not in a TLB, hardware or system software searches for one in the page table following the model defined by the PowerPC architecture.

Instruction and data TLBs provide address translation in parallel with the cache access, incurring no additional time penalty in the event of a TLB hit. The e600 core instruction and data TLBs are 128-entry, two-way set-associative caches that contain address translations. The core can initiate a hardware or system software search of the page tables in memory on a TLB miss.

1.2.3 L1 Instruction and Data Caches Within the Core

The e600 core implements separate L1 instruction and data caches. Each cache is 32 Kbytes and eight-way set-associative. As defined by the PowerPC architecture, they are physically indexed. Each cache block contains eight contiguous words from memory that are loaded from an eight-word boundary (that is, bits EA[27–31] are zeros); thus, a cache block never crosses a page boundary. An entire cache block can be updated by a four-beat burst load across a 64-bit MPX bus. Misaligned accesses across a page boundary can incur a performance penalty. The data cache is a non-blocking, write-back cache with hardware support for reloading on cache misses. The critical double word is transferred on the first beat and is forwarded to the requesting unit, minimizing stalls due to load delays. For vector loads, the critical quad word is handled similarly but is transferred on the second beat. The cache being loaded is not blocked to internal accesses while the load completes.

The e600 core L1 cache organization is shown in [Figure 1-2](#).

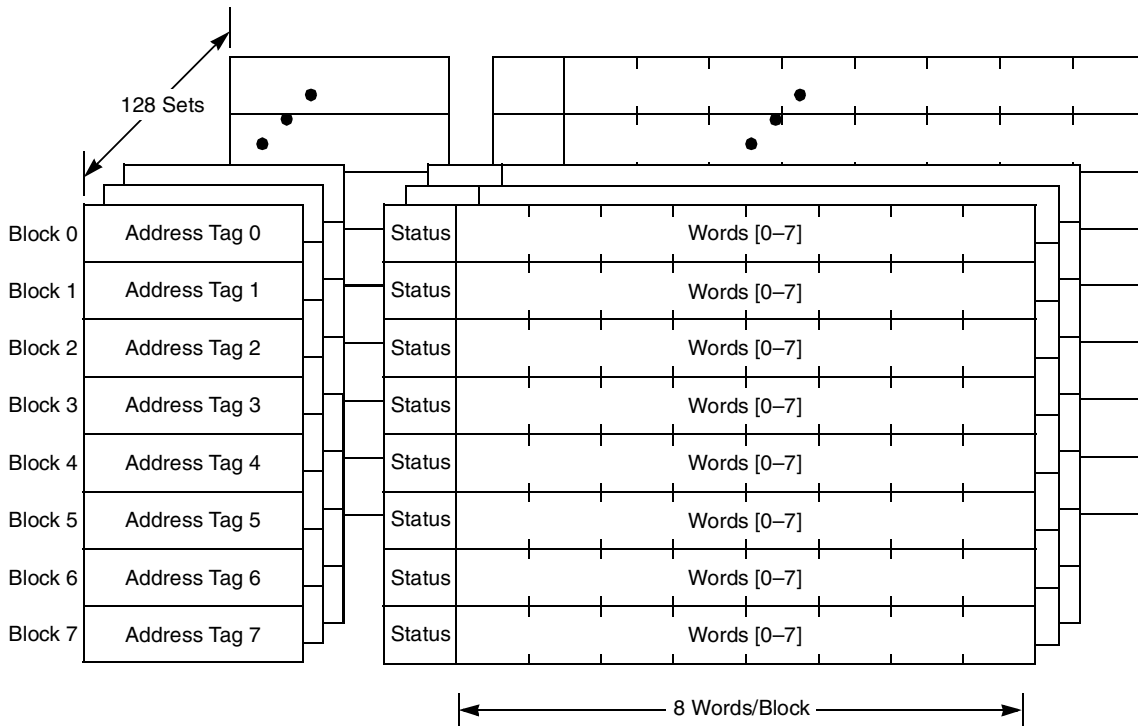


Figure 1-2. L1 Cache Organization

The instruction cache provides up to four instructions per clock cycle to the instruction queue. The instruction cache can be invalidated entirely or on a cache-block basis. It is invalidated and disabled by setting `HID0[ICFI]` and then clearing `HID0[ICE]`. The instruction cache can be locked by setting `HID0[ILOCK]`. The instruction cache supports only the valid/invalid states.

The data cache provides four words per clock cycle to the LSU. Like the instruction cache, the data cache can be invalidated all at once or on a per-cache-block basis. The data cache can be invalidated and disabled by setting `HID0[DCFI]` and then clearing `HID0[DCE]`. The data cache can be locked by setting `HID0[DLOCK]`. The data cache tags are dual-ported, so a load or store can occur simultaneously with a snoop.

The e600 core also implements a 128-entry (32-set, four-way set-associative) branch target instruction cache (BTIC). The BTIC is a cache of branch instructions that have been encountered in branch/loop code sequences. If the target instruction is in the BTIC, it is fetched into the instruction queue a cycle sooner than it can be made available from the instruction cache. Typically, the BTIC contains the first four instructions in the target stream.

The BTIC can be disabled and invalidated through software. As with other aspects of e600 core instruction timing, BTIC operation is optimized for cache-line alignment. If the first target instruction is one of the first five instructions in the cache block, the BTIC entry holds four instructions. If the first target instruction is the last instruction before the cache block boundary, it is the only instruction in the corresponding BTIC entry. If the next-to-last instruction in a cache block is the target, the BTIC entry holds two valid target instructions, as shown in [Figure 1-3](#).

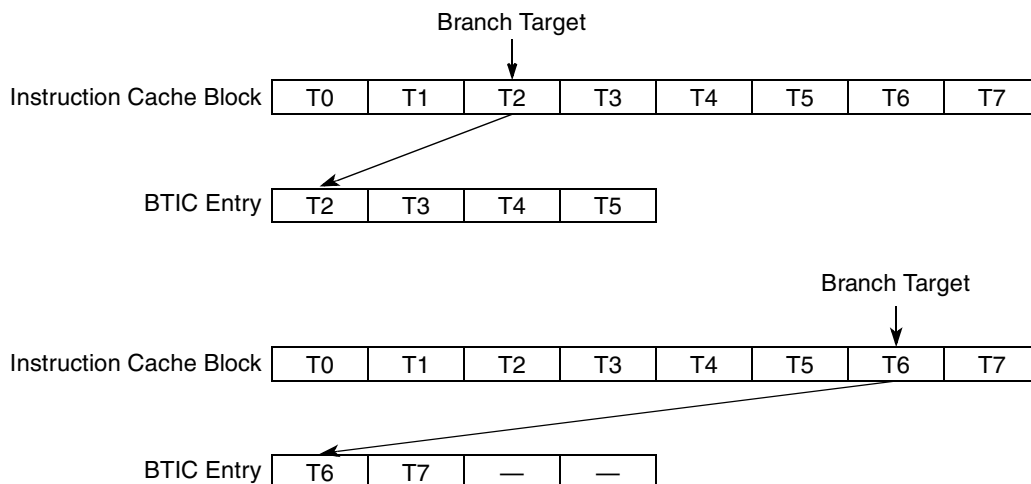


Figure 1-3. Alignment of Target Instructions in the BTIC

BTIC ways are updated using a FIFO algorithm.

For more information and timing examples showing cache hit and cache miss latencies, see [Section 6.3.2, “Instruction Fetch Timing.”](#)

1.2.4 L2 Cache Implementation

The integrated L2 cache is a unified 1-Mbyte cache that receives memory requests from both the L1 instruction and data caches independently. It is eight-way set-associative and organized with 32-byte blocks and two blocks/line.

Each line consists of 64 bytes of data organized as two blocks (also called sectors). Although all 16 words in a cache line share the same address tag, each block maintains the three separate status bits for the 8 words of the cache block, the unit of memory at which coherency is maintained. Thus, each cache line can contain 16 contiguous words from memory that are read or written as 8-word operations.

The integrated L2 cache organization of the e600 core is shown in [Figure 1-4](#).

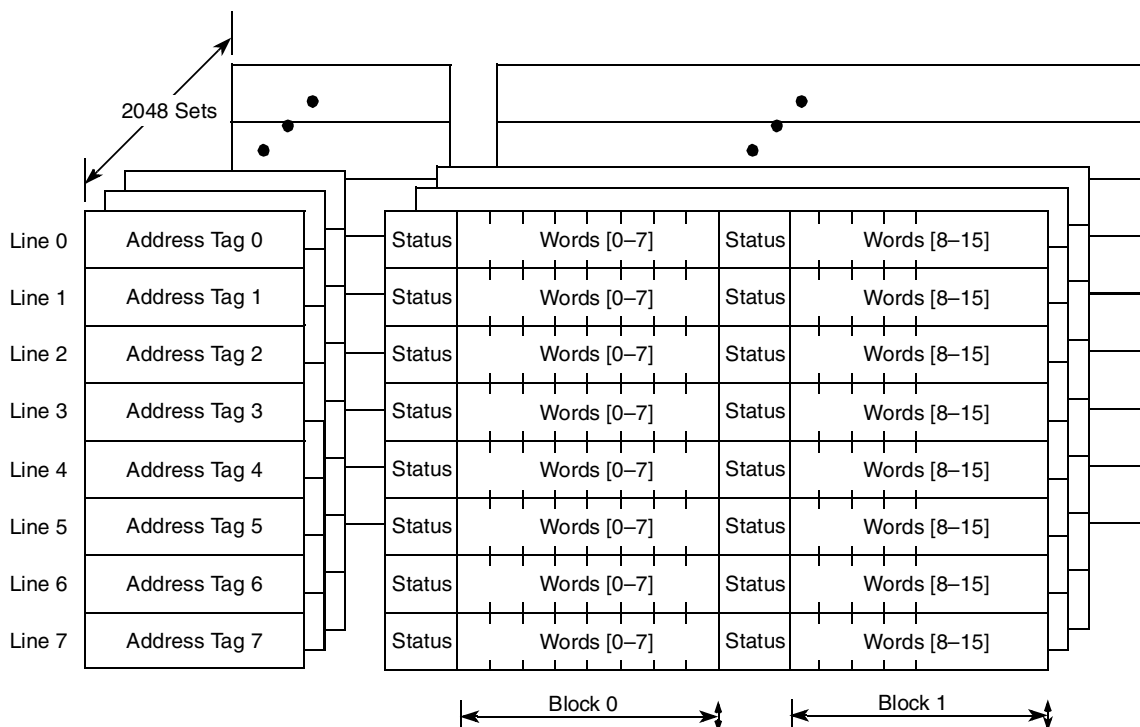


Figure 1-4. L2 Cache Organization

The L2 cache controller contains the L2 cache control register (L2CR), which does the following:

- Includes bits for enabling parity checking on the L2
- Provides for instruction-only and data-only modes
- Provides hardware flushing for the L2
- Selects between two available replacement algorithms for the L2 cache

The L2 implements the MESI cache coherency protocol using three status bits per sector.

Requests from the L1 cache generally result from instruction misses, data load or store misses, write-through operations, or cache management instructions. Requests from the L1 cache are compared against the L2 tags and serviced by the L2 cache if they hit; if they miss in the L2 cache, they go to main memory.

The L2 cache tags are fully pipelined and non-blocking for efficient operation. Thus the L2 cache can be accessed internally while a load for a miss is pending (allowing hits under misses). A reload for a cache miss is treated as a normal access and blocks other accesses for only 1 cycle.

For more information, see [Chapter 3, “L1 and L2 Cache Operation.”](#)

1.2.5 Core Interface

The e600 core has an advanced bus interface, the MPX bus interface. The MPX bus includes a 72-bit data bus (including 8 parity bits) and a 41-bit address bus (including 5 parity bits) along with sufficient control signals to allow for unique system level optimizations. The MPX bus has the following features:

- Extended 36-bit address bus plus 5 bits of odd parity (41 bits total)
- 64-bit data bus plus 8 bits of odd parity (72 bits total); a 32-bit data bus mode is not supported
- Support for a four-state (MESI) cache coherence protocol
- Snooping within the core to maintain L1 data cache, and L2 cache coherency for multiprocessing applications and DMA environments
- Support for address-only transfers (useful for a variety of broadcast operations in multiprocessor applications)
- Address pipelining
- Support for up to 16 out-of-order transactions
- Full data streaming
- Support for data intervention in multiprocessor systems

1.2.6 Overview of Core Interface Accesses

The core interface includes address register queues, prioritization logic, and a bus control unit. The core interface latches snoop addresses for snooping in the L1 data cache and the L2 cache, the memory hierarchy address register queues, and the reservation controlled by the Load Word and Reserve Indexed (**lwarx**) and Store Word Conditional Indexed (**stwcx.**) instructions. Accesses are prioritized with load operations preceding store operations.

Instructions are automatically fetched from the memory system into the instruction unit where they are issued to the execution units at a peak rate of three instructions per clock cycle. Conversely, load and store instructions explicitly specify the movement of operands to and from the integer, floating-point, and AltiVec register files and the memory system.

When the e600 core encounters an instruction or data access, it calculates the effective address and uses the lower-order address bits to check for a hit in the 32-Kbyte L1 instruction and data caches within the core. During L1 cache lookup, the instruction and data memory management units (MMUs) use the higher-order address bits to calculate the virtual address, from which they calculate the physical (real) address. The physical address bits are then compared with the corresponding cache tag bits to determine if a cache hit occurred in the L1 instruction or data cache. If the access misses in the corresponding cache, the transaction is sent to L1 load miss queue or the L1 store miss queue. L1 load miss queue transactions are sent to the internal 1-Mbyte L2 cache. Store miss queue transactions are queued up in the L2 cache controller. If no match is found in the L2 tags, the physical address is used to access system memory.

In addition to loads, stores, and instruction fetches, the e600 core performs hardware table search operations following TLB misses, L1 and L2 cache castout operations, and cache-line snoop push operations when a modified cache line detects a snoop hit from another bus master.

1.2.6.1 Signal Groupings

A subset of the selected internal e600 core signals is grouped as follows:

- Interrupts/resets—These signals include the external interrupt signal, checkstop signals, and both soft reset and hard reset signals. They are used to interrupt and, under various conditions, to reset the core.
- Core status and control—These signals indicate the state of the core. They include the time-base enable, machine quiesce control, and power management signals.
- Clock control—These signals determine the system clock frequency and provide a flexible clocking scheme that allows the processor to operate at an integer multiple of the system clock frequency. They are also used to synchronize multiprocessor systems.
- Test interface—The JTAG (IEEE Std. 1149.1a™-1993) interface and the common on-chip processor (COP) unit provide a serial interface to the core for performing board-level boundary-scan interconnect tests. The test data input (*tdi*) and test data output (*tdo*) scan ports are used to scan instructions as well as data into the various scan registers for JTAG operations. The scan operation is controlled by the test access port (TAP) controller which in turn is controlled by the test mode select (*tms*) input sequence. The scan data is latched in at the rising edge of test clock (*tck*).
- Master address bus—These signals provide information about the type of transfer, such as whether the transaction is bursted, global, write-through, or cache-inhibited. A signal is also provided that indicates if a transfer error occurred.

NOTE

Active-low signals are shown with overbars. For example, \overline{int} (interrupt) and \overline{sreset} (soft reset). Active-low signals are referred to as asserted (active) when they are low and negated when they are high. Signals that are not active low, such as *sysclk* (system clock) and *tdo* (JTAG test data output) are referred to as asserted when they are high and negated when they are low.

Figure 1-5 shows the subset of internal core interface signals. Signal functionality is described in detail in Chapter 8, “Core Interface.”

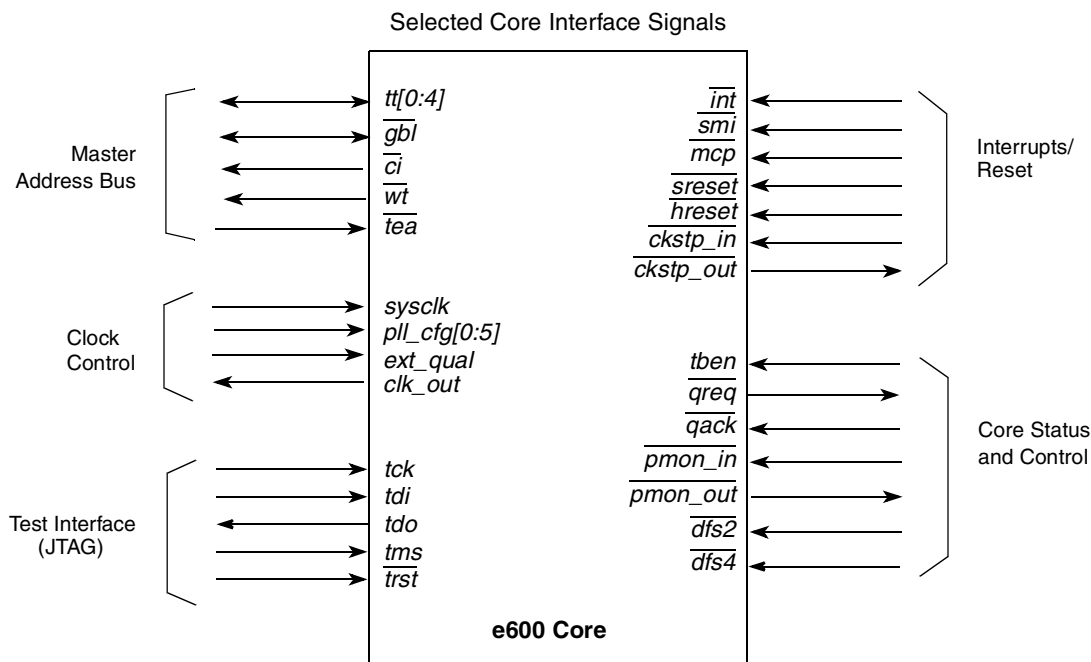


Figure 1-5. Core Interface Signals

1.2.6.2 Clocking

For functional operation, the e600 core uses a single clock input signal, *sysclk*, from which clocking is derived for the processor core and the MPX bus interface. Additionally, internal clock information is made available at the periphery of the core to support debug and development.

The e600 core clocking structure supports a wide range of core-to-MPX bus clock ratios. The internal core clock is synchronized to *sysclk* with the aid of a VCO-based PLL. The *pll_cfg[0:5]* signals are used to program the internal clock rate to a multiple of *sysclk*. The bus clock is maintained at the same frequency as *sysclk*. *sysclk* does not need to be a 50% duty-cycle signal.

1.2.7 Power and Thermal Management

The e600 core is designed for low-power operation. It provides both automatic and program-controlled power reduction modes. If an e600 core functional unit is idle, it automatically goes into a low-power mode. This mode does not affect operational performance. Dynamic power management automatically supplies or withholds power to execution units individually, based upon the contents of the instruction stream. The operation of dynamic power management is transparent to software or any hardware external to the core.

The following three programmable power modes are available to the system:

- Nap—Instruction fetching is halted. Only those clocks for time base, decremter, and JTAG logic remain running. The e600 core goes into the doze state to snoop memory operations on the MPX bus and then back to nap using a *qreq/qack* processor-system handshake protocol.

- Sleep—Power consumption is further reduced by disabling bus snooping, leaving only the PLL in a locked and running state. All internal functional units are disabled.
- Deep sleep—The system can disable the PLL. The system can then disable the SYSCLK source for greater system power savings. Power-on reset procedures for restarting and relocking the PLL must be followed upon exiting deep sleep.

The dynamic frequency switching (DFS) feature conserves power by lowering core operating frequency. The e600 has the ability to divide the core-to-MPX bus ratio by two or four during normal functional operation. [Section 9.6.1, “Dynamic Frequency Switching \(DFS\),”](#) describes power saving with DFS.

The e600 core also provides an instruction cache throttling mechanism to reduce the instruction execution rate. For thermal management, the e600 core provides a supervisor-level instruction cache throttling control register (ICTC). [Chapter 9, “Power and Thermal Management,”](#) explains how to configure the ICTC register for the e600 core.

1.2.8 Core Performance Monitor

The e600 core incorporates a performance monitor facility that system designers can use to help bring up, debug, and optimize software performance. The performance monitor counts events during execution of instructions related to dispatch, execution, completion, and memory accesses.

The performance monitor incorporates several registers that can be read and written to by supervisor-level software. User-level versions of these registers provide read-only access for user-level applications. These registers are described in [Section 1.3.1, “PowerPC Registers and Programming Model.”](#) Performance monitor control registers, MMCR0, MMCR1, and MMCR2 can be used to specify which events are to be counted and the conditions for which a performance monitoring interrupt is taken. Additionally, the sampled instruction address register, SIAR (USIAR), holds the address of the first instruction to complete after the counter overflowed.

Attempting to write to a user-level read-only performance monitor register causes a program interrupt, regardless of the MSR[PR] setting.

When a performance monitor interrupt occurs, program execution continues from vector offset 0x00F00.

[Chapter 10, “Performance Monitor,”](#) describes the operation of the performance monitor diagnostic tool incorporated in the e600 core.

1.3 e600 Core Architectural Implementation

The PowerPC architecture consists of three layers. Adherence to the PowerPC architecture can be described in terms of which of the following levels of the architecture is implemented:

- PowerPC user instruction set architecture (UISA)—Defines the base user-level instruction set, user-level registers, data types, floating-point exception model, memory models for a uniprocessor environment, and programming model for a uniprocessor environment
- PowerPC virtual environment architecture (VEA)—Describes the memory model for a multiprocessor environment, defines cache control instructions, and describes other aspects of virtual environments. Implementations that conform to the VEA also adhere to the UISA, but may not necessarily adhere to the OEA.

- PowerPC operating environment architecture (OEA)—Defines the memory management model, supervisor-level registers, synchronization requirements, and the interrupt model. Implementations that conform to the OEA also adhere to the UISA and the VEA.

The e600 core implementation supports the three levels of the architecture described above. For more information about the PowerPC architecture, see the *Programming Environments Manual for 32-Bit Implementations of the PowerPC Architecture*. Features specific to the e600 core are listed in [Section 1.2, “e600 Core Features.”](#)

This section describes the PowerPC architecture in general, and specific details about the implementation of the e600 core as a low-power, 32-bit device that implements this architecture. The structure of this section follows the reference manual organization; each subsection provides an overview of that chapter.

- Registers and programming model—[Section 1.3.1, “PowerPC Registers and Programming Model,”](#) describes the registers for the operating environment architecture common to the e600 core and describes the programming model. It also describes the registers that are unique to the e600 core.
Instruction set and addressing modes—[Section 1.3.2, “Instruction Set,”](#) describes the PowerPC instruction set and addressing modes for the PowerPC operating environment architecture, and defines and describes the PowerPC instructions implemented in the e600 core. The information in this section is described more fully in [Chapter 2, “Registers.”](#)
- Cache implementation—[Section 1.3.3, “Cache Implementation within the Core,”](#) describes the cache model that is defined generally by the virtual environment architecture. It also provides specific details about the e600 core cache implementation. The information in this section is described more fully in [Chapter 3, “L1 and L2 Cache Operation.”](#)
- Interrupt model—[Section 1.3.4, “Interrupt Model,”](#) describes the interrupt model of the PowerPC operating environment architecture and the differences in the e600 core interrupt model. The information in this section is described more fully in [Chapter 4, “Interrupts.”](#)
- Memory management—[Section 1.3.5, “Memory Management,”](#) describes generally the conventions for memory management. This section also describes the e600 core implementation of the 32-bit PowerPC memory management specification. The information in this section is described more fully in [Chapter 5, “Memory Management.”](#)
- Instruction timing—[Section 1.3.6, “Instruction Timing,”](#) provides a general description of the instruction timing provided by the parallel execution supported by the PowerPC architecture and the e600 core. The information in this section is described more fully in [Chapter 6, “Instruction Timing.”](#)
- AltiVec implementation—[Section 1.3.7, “AltiVec Implementation,”](#) points out that the e600 core implements AltiVec registers, instructions, and interrupts as described in the *AltiVec Technology Programming Environments Manual*. [Chapter 7, “AltiVec Technology Implementation,”](#) provides complete details.

1.3.1 PowerPC Registers and Programming Model

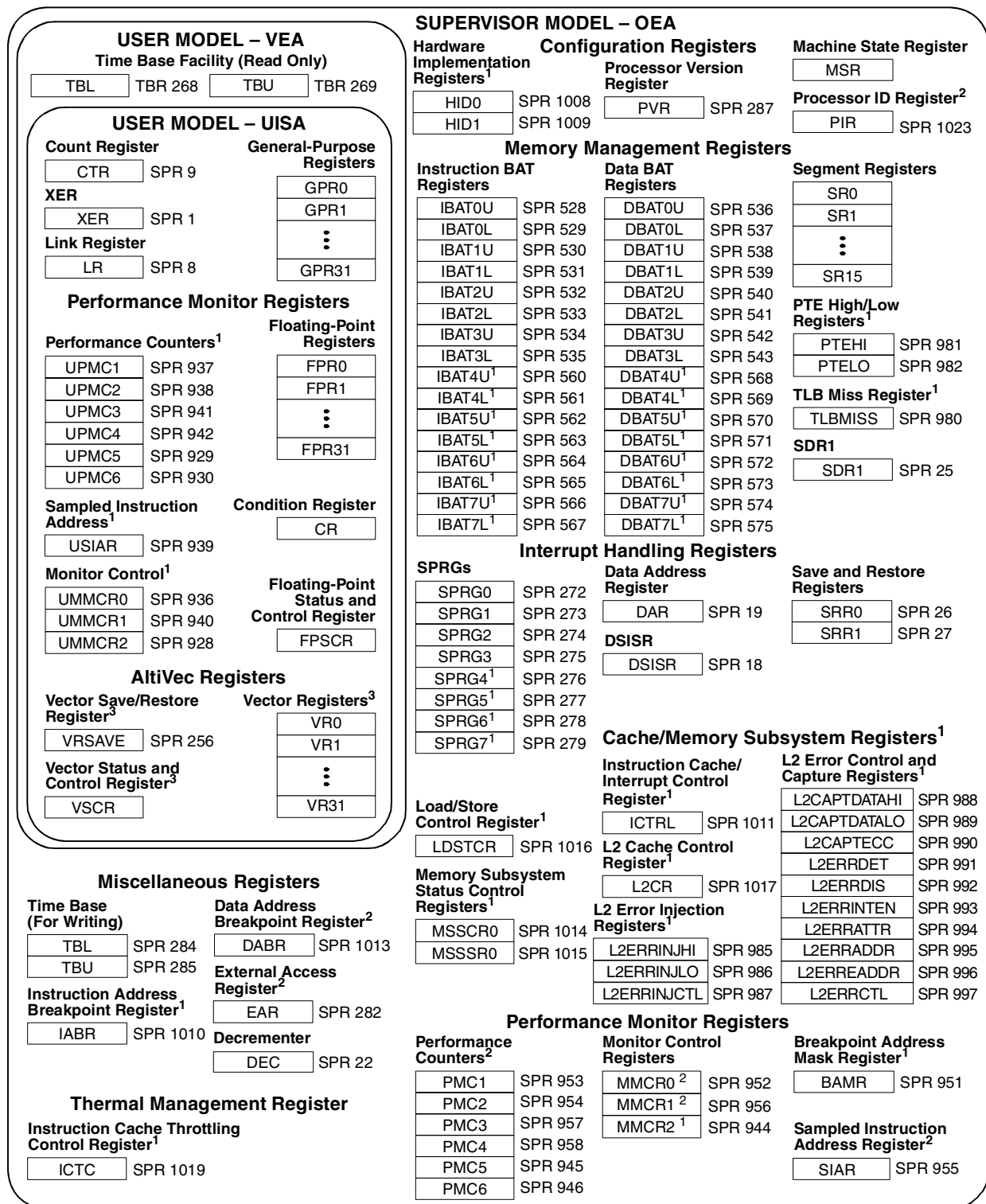
The PowerPC architecture defines register-to-register operations for most computational instructions. Source operands for these instructions are accessed from the registers or are provided as immediate values embedded in the instruction opcode. The three-register instruction format allows specification of a target register distinct from the two source operands. Load and store instructions transfer data between registers and memory.

The PowerPC architecture also defines two levels of privilege—supervisor mode of operation (typically used by the operating system) and user mode of operation (used by the application software). The programming models incorporate 32 GPRs, 32 FPRs, SPRs, and several miscellaneous registers. The AltiVec extensions to the PowerPC architecture augment the programming model with 32 VRs, one status and control register, and one save and restore register. Each processor that implements the PowerPC architecture also has a unique set of implementation-specific registers to support functionality that may not be defined by the PowerPC architecture.

Having access to privileged instructions, registers, and other resources allows the operating system to control the application environment (providing virtual memory and protecting operating-system and critical machine resources). Instructions that control the state of the processor, the address translation mechanism, and supervisor registers can be executed only when the processor is operating in supervisor mode.

[Figure 1-6](#) shows all of the e600 core registers, indicating which are available at the user and supervisor levels. The numbers to the right of the SPRs indicate the number that is used in the syntax of the instruction operands to access the register. For more information, see [Chapter 2, “Registers.”](#)

The OEA defines numerous SPRs that serve a variety of functions, such as providing controls, indicating status, configuring the processor, and performing special operations. During normal execution, a program can access the registers shown in [Figure 1-6](#), depending on the program’s access privilege (supervisor or user, determined by the privilege-level bit, MSR[PR]). GPRs, FPRs, and VRs are accessed through operands that are part of the instructions. Access to registers can be explicit (that is, through the use of specific instructions for that purpose such as Move to Special-Purpose Register (**mtspr**) and Move from Special-Purpose Register (**mfspr**) instructions) or implicit, as the part of the execution of an instruction.



Some registers can be accessed both explicitly and implicitly. In the e600 core, all SPRs are 32 bits wide. [Table 2-1](#) describes the registers implemented by the e600 core.

1.3.2 Instruction Set

All PowerPC instructions are encoded as single-word (32-bit) opcodes. Instruction formats are consistent among all instruction types, permitting efficient decoding to occur in parallel with operand accesses. This fixed instruction length and consistent format greatly simplifies instruction pipelining.

For more information, see [Chapter 2, “Registers.”](#)

1.3.2.1 PowerPC Instruction Set

The PowerPC instructions are divided into the following categories:

- Integer instructions—These include computational and logical instructions.
 - Integer arithmetic instructions
 - Integer compare instructions
 - Integer logical instructions
 - Integer rotate and shift instructions
- Floating-point instructions—These include floating-point computational instructions, as well as instructions that affect the FPSCR.
 - Floating-point arithmetic instructions
 - Floating-point multiply/add instructions
 - Floating-point rounding and conversion instructions
 - Floating-point compare instructions
 - Floating-point status and control instructions
- Load and store instructions—These include integer and floating-point load and store instructions.
 - Integer load and store instructions
 - Integer load and store multiple instructions
 - Floating-point load and store instructions
 - Primitives used to construct atomic memory operations (**lwarx** and **stwex** instructions)
- Flow control instructions—These include branching instructions, condition register logical instructions, trap instructions, and other instructions that affect the instruction flow.
 - Branch and trap instructions
 - Condition register logical instructions
- Processor control instructions—These instructions are used for synchronizing memory accesses and management of caches, TLBs, and the segment registers.
 - Move to/from SPR instructions (**mtspr**, **mfspir**)
 - Move to/from MSR (**mtmsr**, **mfmsr**)
 - Synchronize (**sync**)
 - Instruction synchronize (**isync**)

- Order loads and stores
- Memory control instructions—These instructions provide control of caches, TLBs, and SRs.
 - Supervisor-level cache management instructions
 - User-level cache instructions
 - Segment register manipulation instructions
 - Translation lookaside buffer management instructions

This grouping does not indicate the execution unit that executes a particular instruction or group of instructions.

Integer instructions operate on byte, half-word, and word operands. Floating-point instructions operate on single-precision (one word) and double-precision (one double word) floating-point operands. The PowerPC architecture uses instructions that are four bytes long and word-aligned. It provides for byte, half-word, and word operand loads and stores between memory and a set of 32 GPRs. It also provides for word and double-word operand loads and stores between memory and a set of 32 floating-point registers (FPRs).

Computational instructions do not modify memory. To use a memory operand in a computation and then modify the same or another memory location, the memory contents must be loaded into a register, modified, and then written back to the target location with distinct instructions.

Processors that implement the PowerPC architecture follow the program flow when they are in the normal execution state. However, the flow of instructions can be interrupted directly by the execution of an instruction or by an asynchronous event. Either kind of interrupt may cause one of several components of the system software to be invoked.

Effective address computations for both data and instruction accesses use 32-bit unsigned binary arithmetic. A carry from bit 0 is ignored in 32-bit implementations.

1.3.2.2 AltiVec Instruction Set

The AltiVec instructions are divided into the following categories:

- Vector integer arithmetic instructions—These include arithmetic, logical, compare, rotate, and shift instructions.
- Vector floating-point arithmetic instructions—These include floating-point arithmetic instructions, as well as a discussion on floating-point modes.
- Vector load and store instructions—These include load and store instructions for vector registers. The AltiVec technology defines LRU and transient type instructions that can be used to optimize memory accesses.
 - LRU instructions. The AltiVec architecture specifies that the **lvxl** and **stvxl** instructions differ from other AltiVec load and store instructions in that they leave cache entries in a least-recently-used (LRU) state instead of a most-recently-used state.
 - Transient instructions. The AltiVec architecture describes a difference between static and transient memory accesses. A static memory access should have some reasonable degree of locality and be referenced several times or reused over some reasonably long period of time. A

transient memory reference has poor locality and is likely to be referenced a very few times or over a very short period of time.

The following instructions are interpreted to be transient:

- **dstt** and **dsttt** (transient forms of the two data stream touch instructions)
- **lvxl** and **stvx**
- Vector permutation and formatting instructions—These include pack, unpack, merge, splat, permute, select, and shift instructions.
- Processor control instructions—These instructions are used to read and write from the AltiVec status and control register.
- Memory control instructions—These instructions are used for managing of caches (user level and supervisor level).

1.3.2.3 e600 Core Instruction Set

The e600 core instruction set is defined as follows:

- The e600 core provides hardware support for all 32-bit PowerPC instructions.
- The e600 core implements the following instructions optional to the PowerPC architecture:
 - External Control In Word Indexed (**eciwx**)
 - External Control Out Word Indexed (**ecowx**)
 - Data Cache Block Allocate (**dcba**)
 - Floating Select (**fsel**)
 - Floating Reciprocal Estimate Single-Precision (**fres**)
 - Floating Reciprocal Square Root Estimate (**frsqrte**)
 - Store Floating-Point as Integer Word (**stfiwx**)
 - Load Data TLB Entry (**tlbld**)
 - Load Instruction TLB Entry (**tlbli**)

1.3.3 Cache Implementation within the Core

The following subsections describe the PowerPC architecture’s treatment of cache in general, and the e600-specific implementation, respectively. A detailed description of the e600 core cache implementation is provided in [Chapter 3, “L1 and L2 Cache Operation.”](#)

1.3.3.1 PowerPC Cache Model

The PowerPC architecture does not define hardware aspects of cache implementations. For example, devices that implement the PowerPC architecture can have unified caches, separate L1 instruction and data caches (Harvard architecture), or no cache at all. These devices control the following memory access modes on a page or block basis:

- Write-back/write-through mode
- Caching-inhibited/caching-allowed mode
- Memory coherency required/memory coherency not required mode

The caches are physically addressed, and the data cache can operate in either write-back or write-through mode as specified by the PowerPC architecture.

The PowerPC architecture defines the term ‘cache block’ as the cacheable unit. The VEA and OEA define cache management instructions a programmer can use to affect cache contents.

1.3.3.2 e600 Core Cache Implementation

The BPU contains a 128-entry BTIC that provides immediate access to cached target instructions.

1.3.4 Interrupt Model

The following sections describe the PowerPC interrupt model and the e600 core implementation. A detailed description of the e600 core interrupt model is provided in [Chapter 4, “Interrupts.”](#)

1.3.4.1 PowerPC Interrupt Model

The OEA portion of the PowerPC architecture defines the mechanism by which processors that implement the PowerPC architecture invoke interrupts. Exception conditions may be defined at other levels of the architecture. For example, the UISA defines floating-point exception conditions; the OEA defines the mechanism by which the resulting interrupt is taken.

The PowerPC interrupt mechanism allows the processor to change to supervisor state as a result of unusual conditions arising in the execution of instructions and from signals external to the core, bus errors, or various internal conditions. When interrupts occur, information about the state of the processor is saved to certain registers and the processor begins execution at an address (interrupt vector) predetermined for each interrupt. Processing of interrupts begins in supervisor mode.

Although multiple exception conditions can map to a single interrupt vector, often a more specific condition may be determined by examining a register associated with the interrupt—for example, the DSISR and the floating-point status and control register (FPSCR). Also, software can explicitly enable or disable some exception conditions.

The PowerPC architecture requires that interrupts be taken in program order; therefore, although a particular implementation may recognize exception conditions out of order, they are handled strictly in order with respect to the instruction stream. When an instruction-caused interrupt is recognized, any unexecuted instructions that appear earlier in the instruction stream, including any that have not yet entered the execute state, are required to complete before the interrupt is taken. In addition, if a single instruction encounters multiple exception conditions, the resulting interrupts are taken and handled sequentially. Likewise, asynchronous, precise interrupts are recognized when they occur, but are not handled until all instructions currently in the execute stage successfully complete execution and report their results.

To prevent loss of state information, interrupt handlers must save the information stored in the machine status save/restore registers, SRR0 and SRR1, soon after the interrupt is taken to prevent this information from being lost due to another interrupt event. Because exceptions can occur while an interrupt handler routine is executing, multiple interrupts can become nested. It is the interrupt handler’s responsibility to save the necessary state information if control is to return to the interrupted program.

In many cases, after the interrupt handler handles an interrupt, there is an attempt to execute the instruction that caused the interrupt. Instruction execution continues until the next interrupt is encountered. Recognizing and handling interrupt conditions sequentially guarantees that the machine state is recoverable and processing can resume without losing instruction results.

The following terms are used to describe the stages of interrupt processing: recognition, taken, and handling.

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

The occurrence of IEEE Std.754 floating-point exceptions may not cause an exception to be taken. These IEEE-defined exceptions are referred to as IEEE floating-point exceptions or floating-point exceptions.

1.3.4.2 e600 Core Interrupts

As specified by the PowerPC architecture, interrupts can be either precise or imprecise and either synchronous or asynchronous. Asynchronous interrupts are caused by events external to the core’s execution; synchronous interrupts are caused by instructions.

The types of interrupts are shown in [Table 1-1](#). Note that all interrupts except for the performance monitor, AltiVec unavailable, instruction address breakpoint, system management, AltiVec assist, and the three software table search interrupts are described in Chapter 6, “Interrupts,” in *The Programming Environments Manual*.

Table 1-1. e600 Core Interrupt Classifications

Synchronous/Asynchronous	Precise/Imprecise	Interrupt Types
Asynchronous, nonmaskable	Imprecise	System reset, machine check
Asynchronous, maskable	Precise	External interrupt, system management interrupt, decremter interrupt, performance monitor interrupt
Synchronous	Precise	Instruction-caused interrupts

The interrupt classifications are discussed in greater detail in [Section 4.2, “e600 Core Interrupt Recognition and Priorities.”](#) For a better understanding of how the e600 core implements precise interrupts, see [Chapter 6, “Instruction Timing.”](#) [Table 1-2](#) lists the interrupts implemented in the e600 core and conditions that cause them. [Table 1-2](#) also notes the e600-specific interrupts.

The three software table search interrupts support software page table searching and are enabled by setting `HID0[STEN]`. See [Section 4.6.15, “TLB Miss Interrupts,”](#) and [Chapter 5, “Memory Management.”](#)

Table 1-2. Interrupts and Exception Conditions

Interrupt Type	Vector Offset	Causing Conditions
Reserved	0x00000	—
System reset	0x00100	Assertion of either \overline{hreset} or \overline{sreset} or at power-on reset
Machine check	0x00200	Assertion of \overline{tea} during a data bus transaction, assertion of \overline{mcp} , an address bus parity error on the MPX bus, a data bus parity error on the MPX bus, an L1 instruction cache error, an L1 data cache error, and a core memory subsystem detected error including the following: <ul style="list-style-type: none"> • L2 data parity error • L2 tag parity error • Single-bit and multiple-bit L2 ECC errors MSR[ME] must be set.
DSI	0x00300	As specified in the PowerPC architecture. Also includes the following: <ul style="list-style-type: none"> • A hardware table search due to a TLB miss on load, store, or cache operations results in a page fault • Any load or store to a direct-store segment (SR[T] = 1) • A lwarx or stwcx. instruction to memory with cache-inhibited or write-through memory/cache access attributes.
ISI	0x00400	As specified in the PowerPC architecture
External interrupt	0x00500	MSR[EE] = 1 and \overline{int} is asserted
Alignment	0x00600	<ul style="list-style-type: none"> • A floating-point load/store, stmw, stwcx., lmw, lwarx, eciwx, or ecowx instruction operand is not word-aligned. • A multiple/string load/store operation is attempted in little-endian mode • An operand of a dcbz instruction is on a page that is write-through or cache-inhibited for a virtual mode access. • An attempt to execute a dcbz instruction occurs when the cache is disabled or locked.
Program	0x00700	As specified in the PowerPC architecture
Floating-point unavailable	0x00800	As specified in the PowerPC architecture
Decrementer	0x00900	As defined by the PowerPC architecture, when the msb of the DEC register changes from 0 to 1 and MSR[EE] = 1
Reserved	0x00A00–00BFF	—
System call	0x00C00	Execution of the System Call (sc) instruction
Trace	0x00D00	MSR[SE] = 1 or a branch instruction is completing and MSR[BE] = 1. The e600 core operates as specified in the OEA by taking this interrupt on an isync .

Table 1-2. Interrupts and Exception Conditions (continued)

Interrupt Type	Vector Offset	Causing Conditions
Reserved	0x00E00	The e600 core does not generate an interrupt to this vector. Other processors may use this vector for floating-point assist interrupts.
Reserved	0x00E10–00EFF	—
Performance monitor	0x00F00	The limit specified in $PMCn$ is met and $MMCR0[ENINT] = 1$ (e600-specific)
AltiVec unavailable	0x00F20	Occurs due to an attempt to execute any non-streaming AltiVec instruction when $MSR[VEC] = 0$. This interrupt is not taken for data streaming instructions (dstx , dss , or dssall). (e600-specific)
ITLB miss	0x01000	Caused when $HID0[STEN] = 1$ and the effective address for an instruction fetch cannot be translated by the ITLB (e600-specific).
DTLB miss-on-load	0x01100	Caused when $HID0[STEN] = 1$ and the effective address for a data load operation cannot be translated by the DTLB (e600-specific).
DTLB miss-on-store	0x01200	Caused when $HID0[STEN] = 1$ and the effective address for a data store operation cannot be translated by the DTLB, or when a DTLB hit occurs, and the changed bit in the PTE must be set due to a data store operation (e600-specific).
Instruction address breakpoint	0x01300	$IABR[0-29]$ matches $EA[0-29]$ of the next instruction to complete and $IABR[BE] = 1$ (e600-specific).
System management interrupt	0x01400	$MSR[EE] = 1$ and \overline{smi} are asserted (e600-specific). See section, “System Management Interrupt (0x01400)” of the “Interrupts” chapter of the e600 PowerPC Core Reference Manual.
Reserved	0x01500–015FF	—
AltiVec assist	0x01600	This e600-specific interrupt supports denormalization detection in Java mode as specified in the <i>AltiVec Technology Programming Environments Manual</i> in Chapter 3, “Operand Conventions.”
Reserved	0x01700–02FFF	—

1.3.5 Memory Management

The following subsections describe the memory management features of the PowerPC architecture, and the e600 core implementation, respectively.

1.3.5.1 PowerPC Memory Management Model

The primary function of the MMU in a processor or core that implements the PowerPC architecture is the translation of logical (effective) addresses to physical addresses (referred to as real addresses in the architecture specification) for memory accesses and I/O accesses (I/O accesses are assumed to be memory-mapped). In addition, the MMU provides access protection on a segment, block, or page basis. Note that the e600 core does not implement the optional direct-store facility.

Two general types of memory accesses generated by processors that implement the PowerPC architecture require address translation—instruction accesses and data accesses generated by load and store instructions. In addition, the addresses specified by cache instructions and the optional external control

instructions also require translation. Generally, the address translation mechanism is defined in terms of the segment descriptors and page tables that the processors use to locate the effective-to-physical address mapping for memory accesses. The segment information translates the effective address to an interim virtual address, and the page table information translates the virtual address to a physical address.

The segment descriptors, used to generate the interim virtual addresses, are stored as segment registers on 32-bit implementations (such as the e600 core). In addition, two translation lookaside buffers (TLBs) are implemented on the e600 core to keep recently used page address translations within the core. Although the PowerPC OEA describes one MMU (conceptually), the e600 core hardware maintains separate TLBs and table search resources for instruction and data accesses that can be performed independently (and simultaneously). Therefore, the e600 core is described as having two MMUs, one for instruction accesses (IMMU) and one for data accesses (DMMU).

The block address translation (BAT) mechanism is a software-controlled array that stores the available block address translations within the core. BAT array entries are implemented as pairs of BAT registers that are accessible as supervisor special-purpose registers (SPRs). There are separate instruction and data BAT mechanisms. In the e600 core, they reside in the instruction and data MMUs, respectively.

The MMUs, together with the interrupt processing mechanism, provide the necessary support for the operating system to implement a paged virtual memory environment and for enforcing protection of designated memory areas. [Chapter 4, “Interrupts,”](#) describes how the MSR controls critical MMU functionality.

1.3.5.2 e600 Core Memory Management Implementation

The e600 core implements separate MMUs for instructions and data. It maintains a copy of the segment registers in the instruction MMU; however, read and write accesses to the segment registers (**mfsr** and **mtsr**) are handled through the segment registers in the data MMU. The e600 core MMU is described in [Chapter 5, “Memory Management.”](#)

The e600 core implements the memory management specification of the PowerPC OEA for 32-bit implementations but adds capability for supporting 36-bit physical addressing. Thus, it provides 4 Gbytes of physical address space accessible to supervisor and user programs, with a 4-Kbyte page size and 256-Mbyte segment size. In addition, the e600 core MMUs use an interim virtual address (52 bits) and hashed page tables in the generation of 32- or 36-bit physical addresses (depending on the setting of `HID0[XAEN]`). Processors that implement the PowerPC architecture also have a BAT mechanism for mapping large blocks of memory. Block range from 128 Kbytes to 256 Mbytes and are software programmable.

The e600 core provides table search operations performed in hardware. The 52-bit virtual address is formed and the MMU attempts to fetch the PTE that contains the physical address from the appropriate TLB within the core. If the translation is not found in either the BAT array or in a TLB (that is, a TLB miss occurs), the hardware performs a table search operation (using a hashing function) to search for the PTE. Hardware table searching is the default mode for the e600 core; however, if `HID0[STEN] = 1`, a software table search is performed.

The e600 core also provides support for table search operations performed in software (if `HID0[STEN]` is set). In this case, the `TLBMIS` register saves the effective address of the access that requires a software

table search. The PTEHI and PTELO registers and the **tlbli** and **tlbld** instructions are used in reloading the TLBs during a software table search operation.

The following interrupts support software table searching if HID0[STEN] is set and a TLB miss occurs:

- For an instruction fetch, an ITLB miss interrupt
- For a data load, a DTLB miss-on-load interrupt
- For a data store, a DTLB miss-on-store interrupt

The e600 core implements the optional TLB invalidate entry (**tlbie**) and TLB synchronize (**tlbsync**) instructions that can be used to invalidate TLB entries.

1.3.6 Instruction Timing

This section describes how the e600 core performs operations defined by instructions and reports the results of instruction execution. The e600 core design minimizes average instruction execution latency, which is the number of clock cycles it takes to fetch, decode, dispatch, issue, and execute instructions and make results available for subsequent instructions. Some instructions, such as loads and stores, access memory and require additional clock cycles between the execute phase and the write-back phase. Latencies depend on whether an access is to cacheable or noncacheable memory, whether it hits in the L1 or L2 cache, whether a cache access generates a write back to memory, whether the access causes a snoop hit from another device that generates additional activity, and other conditions that affect memory accesses.

To improve throughput, the e600 core implements pipelining, superscalar instruction issue, branch folding, removal of fall-through branches, three-level speculative branch handling, and multiple execution units that operate independently and in parallel.

As an instruction passes from stage to stage, the subsequent instruction can follow through the stages as the preceding instruction vacates them, allowing several instructions to be processed simultaneously. Although it may take several cycles for an instruction to pass through all the stages, when the pipeline is full, one instruction can complete its work on every clock cycle. [Figure 1-7](#) represents a generic four-stage pipelined execution unit, which when filled has a throughput of one instruction per clock cycle.

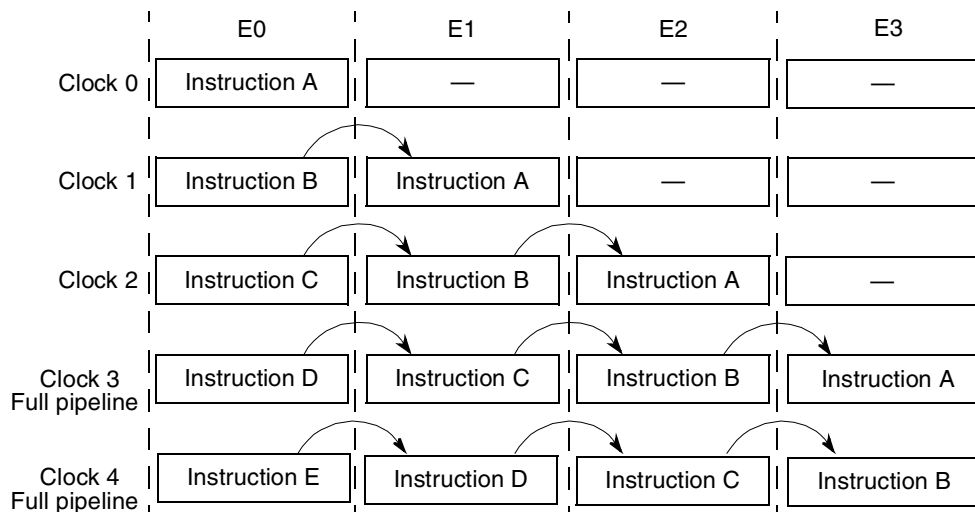


Figure 1-7. Pipelined Execution Unit

Figure 1-8 shows the entire path that instructions take through the fetch1, fetch2, decode/dispatch, execute, issue, complete, and write-back stages, which is considered the master pipeline of the e600 core. The FPU, LSU, IU2, VIU2, VFPU, and VPU are multiple-stage pipelines.

The e600 core contains the following execution units:

- Branch processing unit (BPU)
- Three integer unit 1s (IU1a, IU1b, and IU1c)—execute all integer instructions except multiply, divide, and move to/from SPR instructions.
- Integer unit 2 (IU2)—executes miscellaneous instructions including the CR logical operations, integer multiplication and division instructions, and move to/from special-purpose register instructions
- 64-bit floating-point unit (FPU)
- Load/store unit (LSU)
- The AltiVec unit contains the following four independent execution units for vector computations:
 - AltiVec permute unit (VPU)
 - AltiVec integer unit 1 (VIU1)
 - Vector integer unit 2 (VIU2)
 - Vector floating-point unit (VFPU)

A maximum of two AltiVec instructions can be issued out-of-order to any combination of AltiVec execution units per clock cycle from the bottom two VIQ entries (VIQ1–VIQ0). An instruction in VIQ1 does not have to wait for an instruction in VIQ0 that is waiting for operand availability.

Moreover, the VIU2, VFPU, and VPU are pipelined, so they can operate on multiple instructions.

The e600 core can complete as many as three instructions on each clock cycle. In general, the e600 core processes instructions in seven stages—fetch1, fetch2, decode/dispatch, issue, execute, complete, and write-back, as shown in Figure 1-8. Note that the pipeline example in Figure 6-1 is similar to the four-stage VFPU pipeline in Figure 1-8.

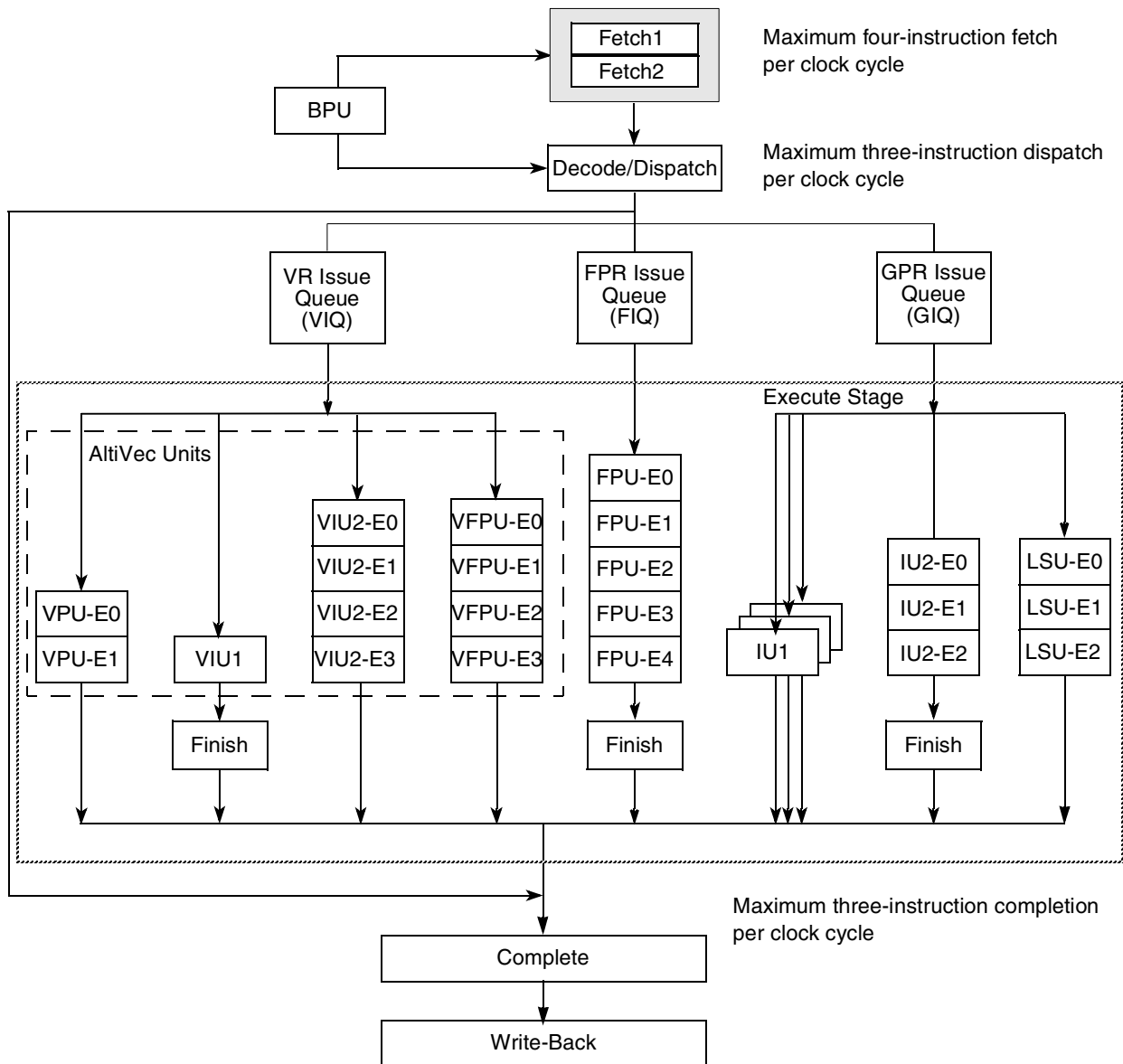


Figure 1-8. Superscalar/Pipeline Diagram

The instruction pipeline stages are described as follows:

- **Instruction fetch**—Includes the clock cycles necessary to request an instruction and the time the memory system takes to respond to the request. Instructions retrieved are latched into the instruction queue (IQ) for subsequent consideration by the dispatcher.

Instruction fetch timing depends on many variables, such as whether an instruction is in the branch target instruction cache (BTIC), the instruction cache, or the L2 cache. Those factors increase when it is necessary to fetch instructions from system memory and include the core-to-MPX bus clock ratio, the amount of bus traffic, and whether any cache coherency operations are required.

- The decode/dispatch stage fully decodes each instruction; most instructions are dispatched to the issue queues (branch, **isync**, **rfi**, and **sc** instructions do not go to issue queues).
- The three issue queues, FIQ, VIQ, and GIQ, can accept as many as one, two, and three instructions, respectively, in a cycle. Instruction dispatch requires the following:
 - Instructions are dispatched only from the three lowest IQ entries—IQ0, IQ1, and IQ2.
 - A maximum of three instructions can be dispatched to the issue queues per clock cycle.
 - Space must be available in the CQ for an instruction to dispatch (this includes instructions that are assigned a space in the CQ but not an issue queue).
- The issue stage reads source operands from rename registers and register files and determines when instructions are latched into the execution unit reservation stations. The GIQ, FIQ, and VIQ (AltiVec) issue queues have the following similarities:
 - Operand lookup in the GPRs, FPRs, and VRs, and their rename registers.
 - Issue queues issue instructions to the proper execution units.
 - Each issue queue holds twice as many instructions as can be dispatched to it in one cycle; the GIQ has six entries, the VIQ has four, and the FIQ has two.

The three issue queues are described as follows:

- The GIQ accepts as many as three instructions from the dispatch unit each cycle. IU1, IU2, and all LSU instructions (including floating-point and AltiVec loads and stores) are dispatched to the GIQ.
- Instructions can be issued out-of-order from the bottom three GIQ entries (GIQ2–GIQ0). An instruction in GIQ1 destined for an IU1 does not have to wait for an instruction in GIQ0 that is stalled behind a long-latency integer divide instruction in the IU2.
- The VIQ accepts as many as two instructions from the dispatch unit each cycle. All AltiVec instructions (other than load, store, and vector touch instructions) are dispatched to the VIQ. A maximum of two AltiVec instructions can be issued out-of-order to any combination of AltiVec execution units per clock cycle from the bottom two VIQ entries (VIQ1–VIQ0). This means an instruction in VIQ1 does not have to wait for an instruction in VIQ0 that is waiting for operand availability.
- The FIQ can accept one instruction from the dispatch unit per clock cycle. It looks at the first instruction in its queue and determines if the instruction can be issued to the FPU in this cycle.
- The execute stage accepts instructions from its issue queue when the appropriate reservation stations are not busy. In this stage, the operands assigned to the execution stage from the issue stage are latched.

The execution unit executes the instruction (perhaps over multiple cycles), writes results on its result bus, and notifies the CQ when the instruction finishes. The execution unit reports any interrupts to the completion stage. Instruction-generated interrupts are not taken until the excepting instruction is next to retire.

Most integer instructions have a 1-cycle latency, so results of these instructions are available 1 clock cycle after an instruction enters the execution unit. The FPU, LSU, IU2, VIU2, VFPU, and VPU units are pipelined, as shown in [Chapter 7, “AltiVec Technology Implementation.”](#)

Note that AltiVec computational instructions are executed in the four independent, pipelined AltiVec execution units. The VPU has a two-stage pipeline, the VIU1 has a one-stage pipeline, and the VIU2 and VFPU have four-stage pipelines. As many as 10 AltiVec instructions can be executing concurrently.

- The complete and write-back stages maintain the correct architectural machine state and commit results to the architected registers in the proper order. If completion logic detects an instruction containing an interrupt status, all following instructions are cancelled, their execution results in rename buffers are discarded, and the correct instruction stream is fetched.

The complete stage ends when the instruction is retired. Three instructions can be retired per clock cycle. If no dependencies exist, as many as three instructions are retired in program order.

[Chapter 6, “Instruction Timing,”](#) describes completion dependencies.

The write-back stage occurs in the clock cycle after the instruction is retired.

1.3.7 AltiVec Implementation

The e600 core implements the AltiVec registers and instruction set as they are described in the *AltiVec Technology Programming Environments Manual* in Chapter 2, “AltiVec Register Set,” and in Chapter 6, “AltiVec Instructions.” One additional implementation-specific interrupt, the AltiVec assist interrupt, is used in handling denormalized numbers in Java mode.

Both interrupts are described fully in [Chapter 4, “Interrupts.”](#) Also, the default setting for the VSCR[NJ] bit is Java-compliant (VSCR[NJ] = 0) in the e600 core. The AltiVec implementation is described fully in [Chapter 7, “AltiVec Technology Implementation.”](#)

Chapter 2 Registers

This chapter describes the PowerPC register model on the e600 core, emphasizing those features specific to the e600 core and summarizing those that are common to processors that implement the PowerPC architecture. It consists of three major sections, which describe the following:

- Registers implemented in the e600 core
- Operand conventions
- The e600 core instruction set

For detailed information about architecture-defined features, see the *Programming Environments Manual* and the *AltiVec Technology Programming Environments Manual*.

AltiVec Technology and the Programming Model

AltiVec programming model features are described as follows:

- Thirty-four additional registers—32 VRs, VRSAVE, and VSCR. See [Section 7.1, “AltiVec Technology and the Programming Model.”](#)

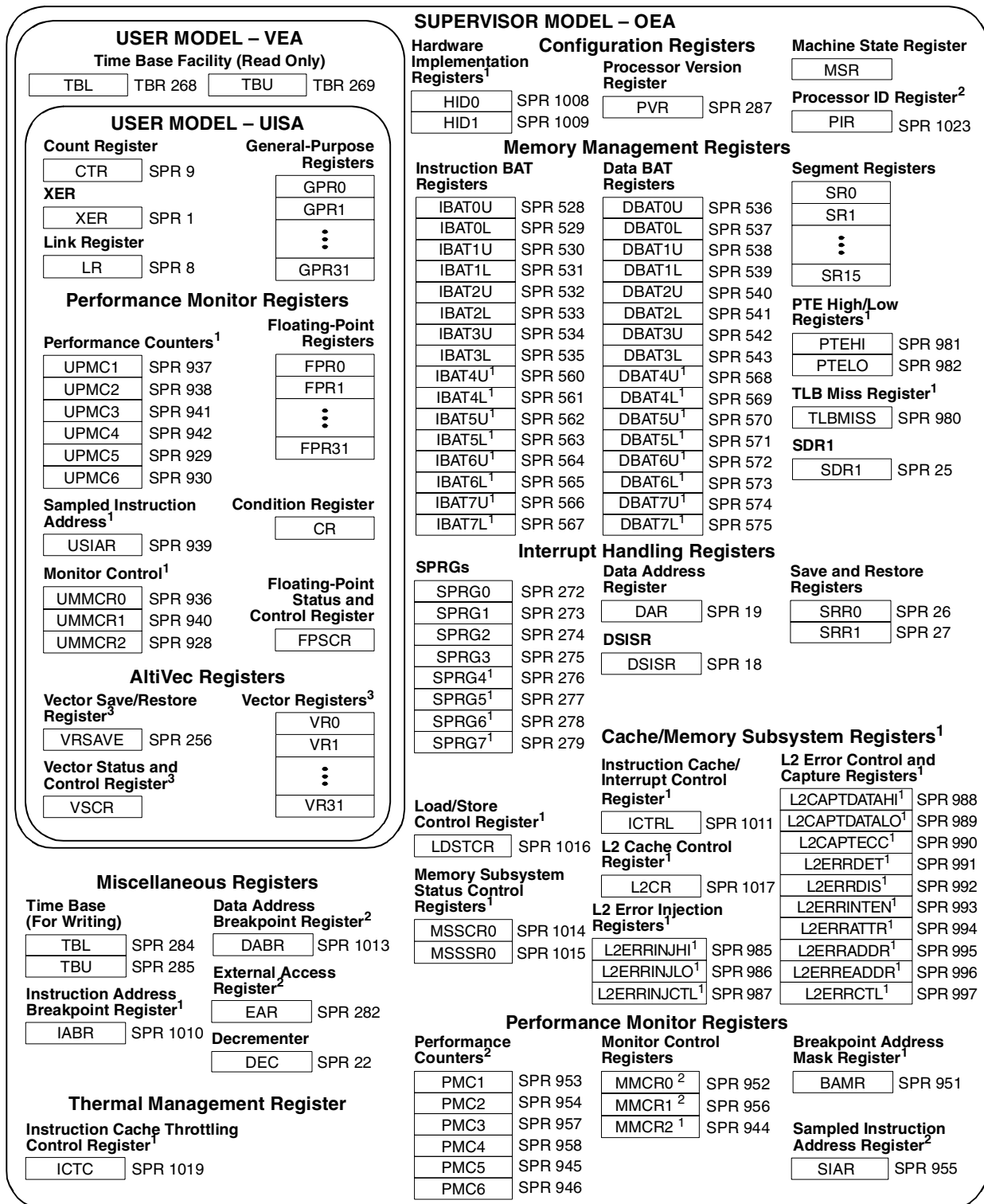
2.1 e600 Core Register Set

This section describes the registers implemented in the e600 core. It includes an overview of registers defined by the PowerPC architecture and the AltiVec technology, highlighting differences in how these registers are implemented in the e600 core, and a detailed description of e600-specific registers. Full descriptions of the architecture-defined register set are provided in Chapter 2, “Register Set,” in the *Programming Environments Manual* and Chapter 2, “AltiVec Register Set,” in the *AltiVec Technology Programming Environments Manual* (PEM).

Registers are defined at all three levels of the PowerPC architecture—user instruction set architecture (UISA), virtual environment architecture (VEA), and operating environment architecture (OEA). The PowerPC architecture defines register-to-register operations for all computational instructions. Source data for these instructions is accessed from the registers within the core or provided as immediate values embedded in the opcode. The three-register instruction format allows specification of a target register distinct from the two source registers, thus preserving the original data for use by other instructions and reducing the number of instructions required for certain operations. Data is transferred between memory and registers with explicit load and store instructions only.

2.1.1 Register Set Overview

[Figure 2-1](#) shows the e600 core register set.



¹ e600-specific register that may not be supported on other processors that implement the PowerPC architecture.
² Register defined as optional in the PowerPC architecture.
³ Register defined by the AltiVec technology.

Figure 2-1. Programming Model—e600 Core Registers

The number to the right of the special-purpose registers (SPRs) is the number used in the syntax of the instruction operands to access the register (for example, the number used to access the XER register is SPR 1). These registers can be accessed using **mtspr** and **mfspr**. Note that not all registers in [Figure 2-1](#) are SPRs, for example VSCR and VRs are AltiVec registers and do not have an SPR number.

2.1.2 e600 Core Register Set

[Table 2-1](#) summarizes the registers implemented in the e600 core.

Table 2-1. e600 Core Register Summary

Name	SPR	Description	Reference/Section
UISA Registers			
CR	—	Condition register. The 32-bit CR consists of eight 4-bit fields, CR0–CR7, that reflect results of certain arithmetic operations and provide a mechanism for testing and branching.	PEM
CTR	9	Count register. Holds a loop count that can be decremented during execution of appropriately coded branch instructions. The CTR can also provide the branch target address for the Branch Conditional to Count Register (bcctrx) instruction.	PEM
FPR0–FPR31	—	Floating-point registers (FPR n). The 32 FPRs serve as the data source or destination for all floating-point instructions.	PEM
FPSCR	—	Floating-point status and control register. Contains floating-point exception signal bits, exception summary bits, exception enable bits, and rounding control bits for compliance with the IEEE 754 standard.	PEM
GPR0–GPR31	—	General-purpose registers (GPR n). The thirty-two GPRs serve as data source or destination registers for integer instructions and provide data for generating addresses.	PEM
LR	8	Link register. Provides the branch target address for the Branch Conditional to Link Register (bclrx) instruction, and can be used to hold the logical address of the instruction that follows a branch and link instruction, typically used for linking to subroutines.	PEM
UMMCR0 ¹ UMMCR1 ¹ UMMCR2 ¹	936 940 928	User monitor mode control registers (UMMCR n). Used to enable various performance monitor interrupt functions. UMMCRs provide user-level read access to MMCR registers.	2.1.6.9.2, 10.3.2.1, 2.1.6.9.4, 10.3.3.1, 2.1.6.9.6, 10.3.4.1
UPMC1– UPMC6 ¹	937, 938 941, 942 929, 930	User performance monitor counter registers (UPMC n). Used to record the number of times a certain event has occurred. UPMCs provide user-level read access to PMC registers.	2.1.6.9.9, 10.3.6.1
USIAR ¹	939	User sampled instruction address register. Contains the effective address of an instruction executing at or around the time that the core signals the performance monitor interrupt. USIAR provides user-level read access to the SIAR.	2.1.6.9.11 10.3.7.1
VR0–VR31 ²	—	Vector registers (VR n). Data source and destination registers for all AltiVec instructions.	7.1.1.3
VRSAVE ²	256	Vector save/restore register. Defined by the AltiVec technology to assist application and operating system software in saving and restoring the architectural state across process context-switched events. The register is maintained only by software to track live or dead information on each AltiVec register.	7.1.1.5

Table 2-1. e600 Core Register Summary (continued)

Name	SPR	Description	Reference/Section
VSCR ²	—	Vector status and control register. A 32-bit vector register that is read and written in a manner similar to the FPSCR.	7.1.1.4
XER	1	Indicates overflows and carries for integer operations. Implementation Note —To emulate the POWER architecture lscbx instruction, XER[16–23] are be read with mfspr [XER] and written with mtspr [XER].	PEM
VEA			
TBL, TBU (For reading)	TBR 268 TBR 269	Time base facility. Consists of two 32-bit registers, time base lower and upper registers (TBL/TBU). TBL (TBR 268) and TBU (TBR 269) can only be read from and not written to. TBU and TBL can be read with the move from time base register (mftb) instruction. Implementation Note —Reading from SPR 284 or 285 using the mftb instruction causes an illegal instruction interrupt.	PEM, 2.1.5.1, 2.3.5.1
OEA			
BAMR ¹	951	Breakpoint address mask register. Used in conjunction with the events that monitor IABR hits. Note: See Table 2-37 for specific synchronization requirements on this register.	2.1.6.9.7 10.3.5
DABR ³	1013	Data address breakpoint register. Optional register implemented in the e600 core and is used to cause a breakpoint interrupt if a specified data address is encountered. See Table 2-37 for specific synchronization requirements on this register.	PEM
DAR	19	Data address register. After a DSI or alignment interrupt, DAR is set to the effective address (EA) generated by the faulting instruction.	PEM
DEC	22	Decrementer register. A 32-bit decrementer counter used with the decrementer interrupt. Implementation Note —In the e600 core, DEC is decremented and the time base increments at 1/4 of the MPX bus clock frequency.	PEM
DSISR	18	DSI source register. Defines the cause of DSI and alignment interrupts.	PEM
EAR	282	External access register. Used with eciwx and ecowx . Note that the EAR and the eciwx and ecowx instructions are optional in the PowerPC architecture. Since eciwx and ecowx are not implemented in the e600 core, the EAR must not be enabled. Otherwise, attempted execution of eciwx/ecowx causes boundedly undefined results. See Table 2-37 for specific synchronization requirements on this register.	PEM
HID0 ¹ HID1 ¹	1008, 1009	Hardware implementation-dependent registers. Control various functions, such as the power management features, and locking, enabling, and invalidating the instruction and data caches. The HID1 includes bits that reflects the state of <i>pll_cfg</i> [0:5] clock signals and control other bus-related functions. See Table 2-37 for specific synchronization requirements on this register.	2.1.6.1, 2.1.6.2
IABR ¹	1010	Instruction address breakpoint register. Used to cause a breakpoint interrupt if a specified instruction address is encountered. See Table 2-37 for specific synchronization requirements on this register.	2.1.6.6

Table 2-1. e600 Core Register Summary (continued)

Name	SPR	Description	Reference/ Section
IBAT0U/L IBAT1U/L IBAT2U/L IBAT3U/L IBAT4U/L ¹ IBAT5U/L ¹ IBAT6U/L ¹ IBAT7U/L ¹ DBAT0U/L DBAT1U/L DBAT2U/L DBAT3U/L DBAT4U/L ¹ DBAT5U/L ¹ DBAT6U/L ¹ DBAT7U/L ¹	528, 529 530, 531 532, 533 534, 535 560, 561 562, 563 564, 565 566, 567 536, 537 538, 539 540, 541 542, 543 568, 569 570, 571 572, 573 574, 575	Block-address translation (BAT) registers. The PowerPC OEA includes an array of block address translation registers that can be used to specify four blocks of instruction space and four blocks of data space. The BAT registers are implemented in pairs: four pairs of instruction BATs (IBAT0U–IBAT3U and IBAT0L–IBAT3L) and four pairs of data BATs (DBAT0U–DBAT3U and DBAT0L–DBAT3L). Sixteen additional BAT registers are available for the e600 core. These registers are enabled by setting HID0[HIGH_BAT_EN]. When HID0[HIGH_BAT_EN] = 1, the 16 additional BAT registers, organized as 4 pairs of instruction BAT registers (IBAT4U–IBAT7U paired with IBAT4L–IBAT7L) and 4 pairs of data BAT registers (DBAT4U–DBAT7U paired with DBAT4L–DBAT7L) are available. Thus, the e600 can define a total of 16 blocks implemented as 32 BAT registers. Because BAT upper and lower words are loaded separately, software must ensure that BAT translations are correct during the time that both BAT entries are being loaded. The e600 core implements IBAT[G]; however, attempting to execute code from an IBAT area with G = 1 causes an ISI interrupt. See Table 2-37 for specific synchronization requirements on this register.	PEM 5.1.3 and 5.3.1
ICTC ¹	1019	Instruction cache throttling control register. Has bits for enabling instruction cache throttling and for controlling the interval at which instructions are fetched. This controls overall junction temperature.	2.1.6.8 9.7
ICTRL ¹	1011	Instruction cache and interrupt control register. Used in configuring interrupts and error reporting for the instruction and data caches. See Table 2-37 for specific synchronization requirements on this register.	2.1.6.5.15
L2CR ¹	1017	L2 cache control register. Includes bits for enabling parity checking, setting the L2 cache size, and flushing and invalidating the L2 cache.	2.1.6.5.1
L2ERRINJHI ¹ L2ERRINJLO ¹ L2ERRINJCTL ¹ L2CAPTDATAHI ¹ L2CAPTDATALO ¹ L2CAPTDATAEC ¹ L2ERRDET ¹ L2ERRDIS ¹ L2ERRINTEN ¹ L2ERRATTR ¹ L2ERRADDR ¹ L2ERREADDR ¹ L2ERRCTL ¹	985 986 987 988 989 990 991 992 993 994 995 996 997	L2 error registers. The L2 cache supports error injection into the L2 data, data ECC or tag, which can be used to test error recovery software by deterministically creating error scenarios. L2ERRINJHI, L2ERRINJLO, and L2ERRINJCTL are error injection registers. The error control and capture registers control the detection and reporting of tag parity and ECC errors.	2.1.6.5.2 2.1.6.5.3 2.1.6.5.4 2.1.6.5.5 2.1.6.5.6 2.1.6.5.7 2.1.6.5.8 2.1.6.5.9 2.1.6.5.10 2.1.6.5.11 2.1.6.5.12 2.1.6.5.13 2.1.6.5.14
LDSTCR ¹	1016	Load/store control register. Controls data L1 cache way-locking. See Table 2-37 for specific synchronization requirements on this register.	2.1.6.5.16
MMCR0 ³ MMCR1 ³ MMCR2 ¹	952 956 944	Monitor mode control registers (MMCR _n). Enable various performance monitor interrupt functions. UMMCR0–UMMCR2 provide user-level read access to these registers.	2.1.6.9.1 2.1.6.9.3 2.1.6.9.5

Table 2-1. e600 Core Register Summary (continued)

Name	SPR	Description	Reference/ Section												
MSR	—	<p>Machine state register. Defines the processor state. The MSR can be modified by the mtmsr, sc, and rfi instructions. It can be read by the mfmsr instruction. When an interrupt is taken, MSR contents are saved to SRR1. See Section 4.3, “Interrupt Processing.” The following bits are optional in the PowerPC architecture. Note that setting MSR[EE] masks decremter and external interrupts and e600-specific system management and performance monitor interrupts. See Table 2-37 for specific synchronization requirements on this register.</p> <table border="1"> <thead> <tr> <th>Bit</th> <th>Name</th> <th>Description</th> </tr> </thead> <tbody> <tr> <td>6</td> <td>VEC</td> <td> <p>AltiVec available. e600 core and AltiVec technology specific; optional to the PowerPC architecture.</p> <p>0 AltiVec technology is disabled.</p> <p>1 AltiVec technology is enabled.</p> <p>Note: When a non-stream AltiVec instruction accesses VRs or the VSCR when VEC = 0 an AltiVec unavailable interrupt is generated. This does not occur for data streaming instructions (dst(t), dstst(t), and dss); the VRs and the VSCR are available to data streaming instructions even if VEC = 0. VRSAVE can be accessed even if VEC = 0.</p> </td> </tr> <tr> <td>13</td> <td>POW</td> <td> <p>Power management enable. e600-specific and optional to the PowerPC architecture.</p> <p>0 Power management is disabled.</p> <p>1 Power management is enabled. The processor can enter a power-saving mode determined by HID0[NAP,SLEEP] when additional conditions are met. See Table 2-7.</p> </td> </tr> <tr> <td>29</td> <td>PMM</td> <td> <p>Performance monitor marked mode. e600-specific and optional to the PowerPC architecture. See Chapter 10, “Performance Monitor.”</p> <p>0 Process is not a marked process.</p> <p>1 Process is a marked process.</p> </td> </tr> </tbody> </table>	Bit	Name	Description	6	VEC	<p>AltiVec available. e600 core and AltiVec technology specific; optional to the PowerPC architecture.</p> <p>0 AltiVec technology is disabled.</p> <p>1 AltiVec technology is enabled.</p> <p>Note: When a non-stream AltiVec instruction accesses VRs or the VSCR when VEC = 0 an AltiVec unavailable interrupt is generated. This does not occur for data streaming instructions (dst(t), dstst(t), and dss); the VRs and the VSCR are available to data streaming instructions even if VEC = 0. VRSAVE can be accessed even if VEC = 0.</p>	13	POW	<p>Power management enable. e600-specific and optional to the PowerPC architecture.</p> <p>0 Power management is disabled.</p> <p>1 Power management is enabled. The processor can enter a power-saving mode determined by HID0[NAP,SLEEP] when additional conditions are met. See Table 2-7.</p>	29	PMM	<p>Performance monitor marked mode. e600-specific and optional to the PowerPC architecture. See Chapter 10, “Performance Monitor.”</p> <p>0 Process is not a marked process.</p> <p>1 Process is a marked process.</p>	PEM, 2.1.4.4, 4.3
Bit	Name	Description													
6	VEC	<p>AltiVec available. e600 core and AltiVec technology specific; optional to the PowerPC architecture.</p> <p>0 AltiVec technology is disabled.</p> <p>1 AltiVec technology is enabled.</p> <p>Note: When a non-stream AltiVec instruction accesses VRs or the VSCR when VEC = 0 an AltiVec unavailable interrupt is generated. This does not occur for data streaming instructions (dst(t), dstst(t), and dss); the VRs and the VSCR are available to data streaming instructions even if VEC = 0. VRSAVE can be accessed even if VEC = 0.</p>													
13	POW	<p>Power management enable. e600-specific and optional to the PowerPC architecture.</p> <p>0 Power management is disabled.</p> <p>1 Power management is enabled. The processor can enter a power-saving mode determined by HID0[NAP,SLEEP] when additional conditions are met. See Table 2-7.</p>													
29	PMM	<p>Performance monitor marked mode. e600-specific and optional to the PowerPC architecture. See Chapter 10, “Performance Monitor.”</p> <p>0 Process is not a marked process.</p> <p>1 Process is a marked process.</p>													
MSSCR0 ¹	1014	Memory subsystem control register. Used to configure and operate many aspects of the core memory subsystem.	2.1.6.3												
MSSSR0 ¹	1015	Memory subsystem status register. Used to configure and operate the parity functions in the L2 cache for the e600 core. See Table 2-37 for specific synchronization requirements on this register.	2.1.6.4												
PIR	1023	Processor identification register. Provided for system use. All 32 bits of the PIR can be written to with the mtspr instruction.	PEM, 2.1.4.3												
PMC1–PMC6 ³	953, 954 957, 958 945, 946	Performance monitor counter registers (PMC <i>n</i>). Used to record the number of times a certain event has occurred. UPMCs provide user-level read access to these registers.	2.1.6.9.8												
PTEHI, PTELO	981, 982	The PTEHI and PTELO registers are used by the tlbld and tlbli instructions to create a TLB entry. When software table searching is enabled (HID0[STEN] = 1), and a TLB miss interrupt occurs, the bits of the page table entry (PTE) for this access are located by software and saved in the PTE registers.	2.1.6.7.2 5.5.5.1.2												
PVR	287	Processor version register. Read-only register that identifies the version (model) and revision level of the processor.	PEM 2.1.4.1												

Table 2-1. e600 Core Register Summary (continued)

Name	SPR	Description	Reference/Section
SDAR, USDAR	—	Sampled data address register. The e600 core does not implement the optional registers (SDAR or the user-level, read-only USDAR register) defined by the PowerPC architecture. Note that in previous processors the SDA and USDA registers could be written to by boot code without causing an interrupt, this is not the case in the e600 core. A mtspr or mfspr SDAR or USDAR instruction causes a program interrupt.	2.1.6.9.12
SDR1	25	Sample data register. Specifies the base address of the page table entry group (PTEG) address used in virtual-to-physical address translation. Implementation Note —The SDR1 register has been modified (with the SDR1[HTABEXT] and SDR1[HTMEXT] fields) for the e600 core to support the extended 36-bit physical address (when HID0[XAEN] = 1). See Table 2-37 for specific synchronization requirements on this register.	PEM, 2.1.4.6 5.5.1
SIAR ³	955	Sampled instruction address register. Contains the effective address of an instruction executing at or around the time that the processor signals the performance monitor exception condition. USIAR provides user-level read access to the SIAR.	2.1.6.9.10 10.3.7
SPRG0– SPRG3 SPRG4– SPRG7 ¹	272–275 276–279	SPRG <i>n</i> . Provided for operating system use. The SPRG4–7 provide additional registers to be used by system software for software table searching.	PEM 5.5.5.1.3
SR0–SR15	—	Segment registers (SR <i>n</i>). Note that the e600 core implements separate instruction and data MMUs. It associates architecture-defined SRs with the data MMU. It reflects SRs values in separate, shadow SRs in the instruction MMU. See Table 2-37 for specific synchronization requirements on this register.	PEM
SRR0 SRR1	26 27	Machine status save/restore registers (SRR <i>n</i>). Used to save the address of the instruction at which execution continues when rfi executes at the end of an interrupt handler routine. SRR1 is used to save machine status on interrupts and to restore machine status when rfi executes. Implementation Note —When a machine check interrupt occurs, the e600 core sets one or more error bits in SRR1. Refer to the individual interrupts for individual SRR1 bit settings.	PEM, 2.1.4.5 4.3
SVR	286	System version register. Read-only register provided for future product compatibility.	2.1.4.2
TBL TBU (For writing)	284 285	Time base. A 64-bit structure (two 32-bit registers) that maintains the time of day and operating interval timers. The TB consists of two registers—time base upper (TBU) and time base lower (TBL). The time base registers can be written to only by supervisor-level software. TBL (SPR 284) and TBU (SPR 285) can only be written to and not read from. TBL and TBU can be written to, with the Move to Special Purpose Register (mtspr) instruction. Implementation Note —Reading from SPR 284 or 285 causes an illegal instruction interrupt. In the e600 core, DEC is decremented and the time base increments at 1/4 of the MPX bus clock frequency.	PEM, 2.1.5.1, 2.3.5.1
TLBMISS ¹	980	The TLBMISS register is automatically loaded when software searching is enabled (HID0[STEN] = 1) and a TLB miss interrupt occurs. Its contents are used by the TLB miss interrupt handlers (the software table search routines) to start the search process.	2.1.6.7.1 5.5.5.1.1

¹ e600-specific register that may not be supported on other cores or processors that implement the PowerPC architecture.

² Register is defined by the AltiVec technology.

³ Defined as optional register in the PowerPC architecture.

2.1.3 PowerPC User-Level Registers (UISA)

The PowerPC UISA registers are user-level. General-purpose registers (GPRs), floating-point registers (FPRs) and vector registers (VRs) are accessed through instruction operands. Access to registers can be explicit (by using instructions for that purpose such as Move to Special-Purpose Register (**mtspr**) and Move from Special-Purpose Register (**mfspr**) instructions) or implicit as part of the execution of an instruction. Some registers are accessed both explicitly and implicitly.

Implementation Note—The e600 core fully decodes the SPR field of the instruction. If the SPR specified is undefined, an illegal instruction program interrupt occurs.

2.1.4 PowerPC Supervisor-Level Registers (OEA)

The OEA defines the registers an operating system uses for memory management, configuration, interrupt handling, and other operating system functions and they are summarized in [Table 2-1](#). The following supervisor-level register defined by the PowerPC architecture contains additional implementation-specific information for the e600 core.

2.1.4.1 Processor Version Register (PVR)

The PVR identifies the version (model) and revision level of the processor. For more information, see “Processor Version Register (PVR),” in Chapter 2, “PowerPC Register Set,” of the *Programming Environments Manual*. [Table 2-2](#) shows the PVR settings. The revision level is updated for each silicon revision.

Table 2-2. PVR Settings

Device Name	Version No.	Starting Processor Revision Level
MPC7448 1.0	0x8004	0100
MPC8641	0x8004	0010
MPC8641D	0x8004	0010
MPC8610	TBD	TBD

[Table 2-3](#) describes the e600 core PVR bits that are not required by the PowerPC architecture.

Table 2-3. Additional PVR Bits

Bits	Name	Description
0–15	Type	Processor type
16–19	Tech	Processor technology
20–23	Major	Major revision number
24–31	Minor	Minor revision number

2.1.4.2 System Version Register (SVR)

2.1.4.3 Processor Identification Register (PIR)

For more information, see “Processor Identification Register (PIR),” in Chapter 2, “PowerPC Register Set,” of the *Programming Environments Manual*.

Implementation Note—The e600 core provides write access to the PIR with **mtspr** using SPR 1023.

2.1.4.4 Machine State Register (MSR)

The MSR defines the state of the processor. When an interrupt occurs, MSR bits, as described in [Table 2-4](#) are altered as determined by the interrupts. The MSR can also be modified by the **mtmsr**, **sc**, and **rfi** instructions. It can be read by the **mfmsr** instruction.

The MSR is shown in [Figure 2-2](#).



Figure 2-2. Machine State Register (MSR)

The MSR bits are defined in [Table 2-4](#).

Table 2-4. MSR Bit Settings

Bits	Name	Description
0–5	—	Reserved
6	VEC ^{1, 2}	AltiVec vector unit available 0 The processor prevents dispatch of AltiVec instructions (excluding the data streaming instructions— dst , dstt , dstst , dststt , dss , and dssall). The processor also prevents access to the vector register file (VRF) and the vector status and control register (VSCR). Any attempt to execute an AltiVec instruction that accesses the VRF or VSCR, excluding the data streaming instructions generates the AltiVec unavailable interrupt. The data streaming instructions are not affected by this bit; the VRF and VSCR registers are available to the data streaming instructions even when the MSR[VEC] is cleared. 1 The processor can execute AltiVec instructions and the VRF and VSCR registers are accessible to all AltiVec instructions. Note that the VRSAVE register is not protected by MSR[VEC].
7–12	—	Reserved
13	POW ^{1, 3}	Power management enable 0 Power management disabled (normal operation mode) 1 Power management enabled (reduced power mode) Power management functions are implementation-dependent. See Chapter 9, “Power and Thermal Management.”
14	—	Reserved. Implementation-specific
15	ILE	Interrupt little-endian mode. When an interrupt occurs, this bit is copied into MSR[LE] to select the endian mode for the context established by the interrupt.

Table 2-4. MSR Bit Settings (continued)

Bits	Name	Description
16	EE	External interrupt enable 0 The processor delays recognition of interrupts external to the core and decremter interrupt conditions. 1 The processor is enabled to take an interrupt external to the core or the decremter interrupt.
17	PR ⁴	Privilege level 0 The processor can execute both user- and supervisor-level instructions. 1 The processor can only execute user-level instructions.
18	FP ²	Floating-point available 0 The processor prevents dispatch of floating-point instructions, including floating-point loads, stores, and moves. 1 The processor can execute floating-point instructions and can take floating-point enabled program interrupts.
19	ME	Machine check enable 0 Machine check interrupts are disabled. 1 Machine check interrupts are enabled.
20	FE0 ²	IEEE Std. 754 floating-point exception mode 0 (see Table 2-5)
21	SE	Single-step trace enable 0 The processor executes instructions normally. 1 The processor generates a single-step trace interrupt upon the successful execution of every instruction except rfi and sc . Successful execution means that the instruction caused no other interrupt.
22	BE	Branch trace enable 0 The processor executes branch instructions normally. 1 The processor generates a branch type trace interrupt when a branch instruction executes successfully.
23	FE1 ²	IEEE Std. 754 floating-point exception mode 1 (see Table 2-5)
24	—	Reserved. This bit corresponds to the AL bit of the POWER architecture.
25	IP	Interrupt prefix. The setting of this bit specifies whether an interrupt vector offset is prepended with Fs or 0s. In the following description, <i>nnnn</i> is the offset of the interrupt. 0 Interrupts are vectored to the physical address 0x000n_nnnn. 1 Interrupts are vectored to the physical address 0xFFFFn_nnnn.
26	IR ⁵	Instruction address translation 0 Instruction address translation is disabled. 1 Instruction address translation is enabled. For more information see Chapter 5, “Memory Management.”
27	DR ⁴	Data address translation 0 Data address translation is disabled. 1 Data address translation is enabled. For more information see Chapter 5, “Memory Management.”
28	—	Reserved

Table 2-4. MSR Bit Settings (continued)

Bits	Name	Description
29	PMM ¹	Performance monitor marked mode 0 Process is not a marked process. 1 Process is a marked process. This bit can be set when statistics need to be gathered on a specific (marked) process. The statistics will only be gathered when the marked process is executing. e600-specific; defined as optional by the PowerPC architecture. For more information about the performance monitor marked mode bit, see Section 10.4, “Event Counting.”
30	RI	Indicates whether system reset or machine check interrupt is recoverable. indicates whether from the perspective of the processor, it is safe to continue (that is, processor state data such as that saved to SRR0 is valid), but it does not guarantee that the interrupted process is recoverable. 0 Interrupt is not recoverable. 1 Interrupt is recoverable.
31	LE ⁶	Little-endian mode enable 0 The processor runs in big-endian mode. 1 The processor runs in little-endian mode.

¹ Optional to the PowerPC architecture.

² A context synchronizing instruction must follow an mtmsr instruction.

³ A dssall and sync must precede an mtmsr instruction and then a context synchronizing instruction must follow.

⁴ A dssall and sync must precede an mtmsr and then a sync and context synchronizing instruction must follow. Note that if a user is not using the AltiVec data streaming instructions, then a dssall is not necessary prior to accessing the MSR[DR] or MSR[PR] bit.

⁵ A context synchronizing instruction must follow an mtmsr. When changing the MSR[IR] bit the context synchronizing instruction must reside at both the untranslated and the translated address following the mtmsr.

⁶ A dssall and sync must precede an rfi to guarantee a solid context boundary. Note that if a user is not using AltiVec data streaming instructions, a dssall is not necessary before accessing MSR[LE].

Note that setting MSR[EE] masks not only the architecture-defined interrupt and decremter interrupts external to the core but also the e600-specific system management, and performance monitor interrupts.

The IEEE Std. 754 floating-point exception mode bits (FE0 and FE1) together define whether floating-point exceptions are handled precisely, imprecisely, or whether they are taken at all. As shown in [Table 2-5](#), if either FE0 or FE1 are set, the e600 core treats interrupts as precise. MSR bits are guaranteed to be written to SRR1 when the first instruction of the interrupt handler is encountered. For further details, see Chapter 2, “Register Set” and Chapter 6, “Interrupts,” of the *Programming Environments Manual*.

Table 2-5. IEEE Std. 754 Floating-Point Exception Mode Bits

FE0	FE1	Mode
0	0	Floating-point exceptions disabled
0	1	Imprecise nonrecoverable. For this setting, the e600 core operates in floating-point precise mode.
1	0	Imprecise recoverable. For this setting, the e600 core operates in floating-point precise mode.
1	1	Floating-point precise mode

2.1.4.5 Machine Status Save/Restore Registers (SRR0, SRR1)

When an interrupt is taken, the processor uses SRR0 and SRR1 to save the contents of the MSR for the current context and to identify where instruction execution should resume after the interrupt is handled.

When an interrupt occurs, the address saved in SRR0 helps determine where instruction processing should resume when the interrupt handler returns control to the interrupted process. Depending on the interrupt, this may be the address in SRR0 or at the next address in the program flow. All instructions in the program flow preceding this one will have completed execution and no subsequent instruction will have begun execution. This may be the address of the instruction that caused the interrupt or the next one (as in the case of a system call or trace interrupt). The SRR0 register is shown in [Figure 2-3](#).

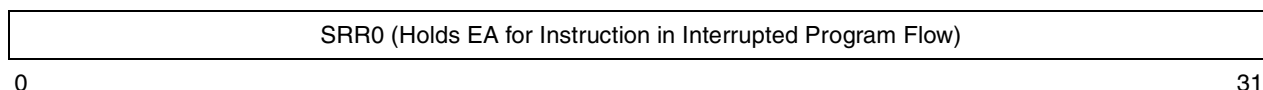


Figure 2-3. Machine Status Save/Restore Register 0 (SRR0)

SRR1 is used to save machine status (selected MSR bits and possibly other status bits) on interrupts and to restore those values when an `rfi` instruction is executed. SRR1 is shown in [Figure 2-4](#).

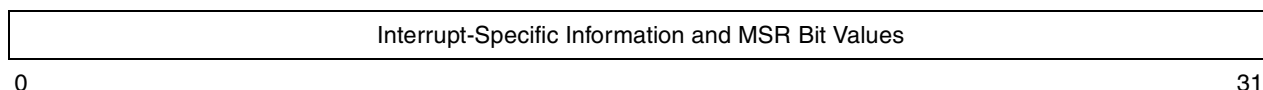


Figure 2-4. Machine Status Save/Restore Register 1 (SRR1)

Typically, when an interrupt occurs, SRR1[0–15] are loaded with interrupt-specific information and MSR[16–31] are placed into the corresponding bit positions of SRR1. For most interrupts, SRR1[0–5] and SRR1[7–15] are cleared, and MSR[6, 16–31] are placed into the corresponding bit positions of SRR1. [Table 2-4](#) provides a summary of the SRR1 bit settings when a machine check interrupt occurs. For a specific interrupt’s SRR1 bit settings, see [Section 4.6, “Interrupt Definitions.”](#)

2.1.4.6 SDR1 Register

The SDR1 register specifies the page table entry group (PTEG) address used in virtual-to-physical address translation. See “SDR1,” in Chapter 2, “PowerPC Register Set,” of the *Programming Environments Manual* for the description with a 32-bit physical address. The SDR1 register has been modified for the e600 core to support the extended 36-bit physical address (when `HID0[XAEN] = 1`). See [Section 5.5.1, “SDR1 Register Definition—Extended Addressing,”](#) for details on how SDR1 is modified to support a 36-bit physical address.

Implementation Note—SDR1[HTABEXT] and SDR1[HTMEXT] fields have been added to support extended addressing. [Section 5.5.1, “SDR1 Register Definition—Extended Addressing,”](#) describes in detail the differences when generating a 36-bit PTEG address. [Figure 2-5](#) shows the format of the modified SDR1.

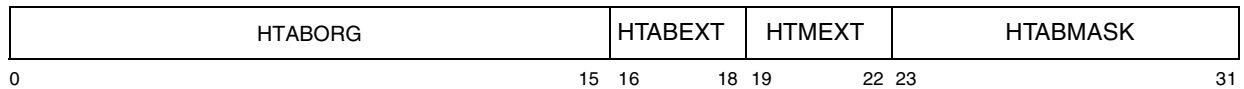


Figure 2-5. SDR1 Register Format—Extended Addressing

Bit settings for the SDR1 register are described in [Table 2-6](#).

Table 2-6. SDR1 Register Bit Settings—Extended Addressing

Bits	Name	Description
0–15	HTABORG	Physical base address of page table If $HID0[XAEN] = 1$, field contains physical address [4–19] If $HID0[XAEN] = 0$, field contains physical address [0–15]
16–18	HTABEXT	Extension bits for physical base address of page table If $HID0[XAEN] = 1$, field contains physical address [1–3] If $HID0[XAEN] = 0$, field is reserved
19–22	HTMEXT	Hash table mask extension bits If $HID0[XAEN] = 1$, field contains hash table mask [0–3] If $HID0[XAEN] = 0$, field is reserved
23–31	HTABMASK	Mask for page table address If $HID0[XAEN] = 1$, field contains hash table mask [4–12] If $HID0[XAEN] = 0$, field contains hash table mask [0–7]

SDR1 can be accessed with **mtspr** and **mfspir** using SPR 25. For synchronization requirements on accesses to the register see [Section 2.3.2.4, “Synchronization.”](#)

2.1.5 PowerPC User-Level Registers (VEA)

The PowerPC VEA defines the time base facility (TB), which consists of two 32-bit registers—time base upper (TBU) and time base lower (TBL).

2.1.5.1 Time Base Registers (TBL, TBU)

The time base registers can be written only by supervisor-level instructions but can be read by both user- and supervisor-level software. The time base registers have two different addresses. TBU and TBL can be read from the TBR 268 and 269, respectively, with the move from time base register (**mftb**) instruction. TBU and TBL can be written to TBR 284 and 285, respectively, with the move to special purpose register (**mtspr**) instruction. Reading from SPR 284 or 285 causes an illegal instruction interrupt. For more information, see “PowerPC VEA Register Set—Time Base,” in Chapter 2, “PowerPC Register Set,” of the *Programming Environments Manual*. Note that DEC is decremented and the time base increments at 1/4 of the MPX bus clock frequency.

2.1.6 e600-Specific Register Descriptions

The PowerPC architecture allows for implementation-specific SPRs. This section describes registers that are defined for the e600 core but are not included in the PowerPC architecture. Note that in the e600 core,

these registers are all supervisor-level registers. All the registers described in the *AltiVec Technology Programming Environments Manual* are implemented in e600 core. See Chapter 2, “AltiVec Register Set,” in the *AltiVec Technology Programming Environments Manual* for details about these registers.

Note that while it is not guaranteed that the implementation of e600-specific registers is consistent among processors that implement the PowerPC architecture, other processors can implement similar or identical registers.

The registers in the following subsections are presented in the order of the chapters in this book. First, the processor control registers are described followed by the cache control registers. Next, the implementation-specific registers for interrupt processing and memory management are presented, followed by the thermal management register. Finally the performance monitor registers are presented.

2.1.6.1 Hardware Implementation-Dependent Register 0 (HID0)

The hardware implementation-dependent register 0 (HID0) controls the state of several functions within the e600 core. The HID0 register is shown in [Figure 2-6](#).

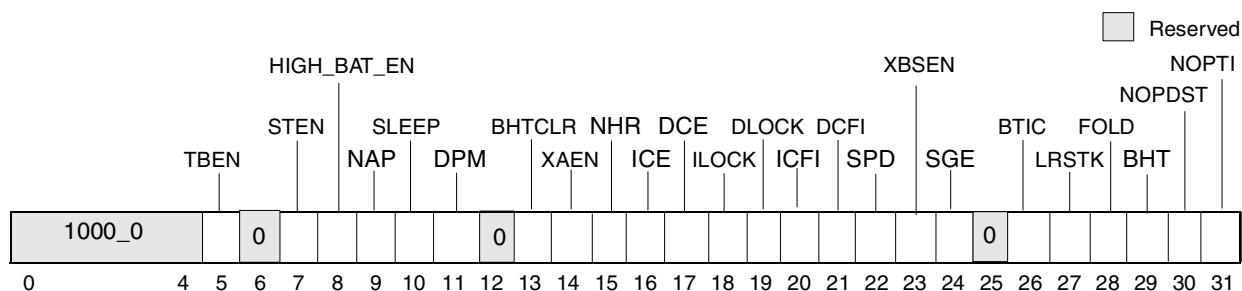


Figure 2-6. Hardware Implementation-Dependent Register 0 (HID0)

The HID0 bits are described in [Table 2-7](#).

Table 2-7. HID0 Field Descriptions

Bits	Name	Description
0–4	—	Reserved. Defined as HID0[0]: EMCP, HID0[2]: EBA, HID0[3]: EBD, HID0[4]: BCLK on some earlier processors. Read as 0b1000_0.
5	TBEN ¹	Time base enable. Note that this bit must be set and the <i>tben</i> signal must be asserted to enable the time base and decremter.
6	—	Reserved.
7	STEN ²	Software table search enable. When a TLB miss occurs, the e600 core takes one of three TLB miss interrupts so that software can search the page tables for the desired PTE. See Section 4.6.15, “TLB Miss Interrupts,” for details on the e600 core facilities for software table searching. 0 Hardware table search enabled 1 Software tables search enabled
8	HIGH_BAT_EN	Additional BATs enabled for the e600 core 0 Additional 4 IBATs (4–7) and 4 DBATs (4–7) disabled 1 Additional 4 IBATs (4–7) and 4 DBATs (4–7) enabled The additional BATs provide for more mapping of memory with the block address translation method.

Table 2-7. HID0 Field Descriptions (continued)

Bits	Name	Description
9	NAP ¹	Nap mode enable. Operates in conjunction with MSR[POW]. 0 Nap mode disabled. 1 Nap mode enabled. Nap mode is invoked by setting MSR[POW] while this bit is set. In nap mode, the PLL and the time base remain active. Note that if both NAP and SLEEP are set, the e600 core ignores the SLEEP bit.
10	SLEEP ¹	Sleep mode enable. Operates in conjunction with MSR[POW]. 0 Sleep mode disabled. 1 Sleep mode enabled. Sleep mode is invoked by setting MSR[POW] while this bit is set. \overline{qreq} is asserted to indicate that the processor is ready to enter sleep mode. If the system logic determines that the core can enter sleep mode, the quiesce acknowledge signal, \overline{qack} , is asserted back to the core. When the \overline{qack} signal assertion is detected, the core enters sleep mode after several core clocks. At this point, the system logic can turn off the PLL by first configuring <i>pll_cfg</i> [0:5] to PLL bypass mode, and then disabling <i>sysclk</i> .
11	DPM ¹	Dynamic power management enable 0 Dynamic power management is disabled. 1 Functional units enter a low-power mode automatically if the unit is idle. This does not affect operational performance and is transparent to software or any hardware external to the core.
12	—	Reserved. For test use; software should not set this bit.
13	BHTCLR ³	Clear branch history table 0 The e600 core clears this bit one cycle after it is set. 1 Setting BHTCLR bit initializes all entries in BHT to weakly, not taken whether or not the BHT is enabled by HID0[BHT]. However, for correct results, the BHT should be disabled (HID0[BHT] = 0) before setting BHTCLR. Setting BHTCLR causes the branch unit to be busy for 64 cycles while the initialization process is completed.
14	XAEN ⁴	Extended addressing enabled 0 Extended addressing is disabled; the 4 most significant bits of the 36-bit physical address are cleared and a 32-bit physical address is used. 1 Extended addressing is enabled; the 32-bit effective address is translated to a 36-bit physical address. If HID0[XAEN] is changed (cleared or set), the BATs and TLBs must be invalidated first.
15	NHR ¹	Not hard reset (software-use only). Helps software distinguish a hard reset from a soft reset. 0 A hard reset occurred if software had previously set this bit. 1 A hard reset has not occurred. If software sets this bit after a hard reset, when a reset occurs and this bit remains set, software knows it was a soft reset. The e600 core never writes this bit unless executing an mtspr (HID0).
16	ICE ⁵	Instruction cache enable 0 The instruction cache is neither accessed nor updated. All pages are accessed as if they were marked cache-inhibited (WIMG = x1xx). Potential cache accesses from the MPX bus (snoop and cache operations) are ignored. In the disabled state for the L1 caches, the cache tag state bits are ignored and all accesses are propagated to the MPX bus as burst transactions. For those transactions, \overline{ci} is asserted regardless of address translation. ICE is zero at power-up. 1 The instruction cache is enabled. Note that HID0[ICFI] must be set at the same time that this bit is set.

Table 2-7. HID0 Field Descriptions (continued)

Bits	Name	Description
17	DCE ²	<p>Data cache enable</p> <p>0 The data cache is neither accessed nor updated. All pages are accessed as if they were marked cache-inhibited (WIMG = x1xx). Potential cache accesses from the MPX bus (snoop and cache operations) are ignored. In the disabled state for the L1 caches, the cache tag state bits are ignored and all accesses are propagated to the L2 cache or MPX bus as cache-inhibited. For those transactions, \overline{ci} is asserted regardless of address translation. DCE is zero at power-up.</p> <p>1 The data cache is enabled. Note that HID0[DCFI] must be set at the same time that this bit is set.</p>
18	ILOCK ⁶	<p>Instruction cache lock</p> <p>0 Normal operation</p> <p>1 All of the ways of the instruction cache are locked. A locked cache supplies data normally on a read hit. On a miss, the access is treated the same as if the instruction cache was disabled. Thus, the MPX bus request is a 32-byte burst read, but the cache is not loaded with data. The data is reloaded into the L2, unless the L2CR[L2DO] bit is set. Note that setting this bit has the same effect as setting ICTRL[ICWL] to all ones. However, when this bit is set, ICTRL[ICWL] is ignored. Chapter 3, “L1 and L2 Cache Operation,” gives further details.</p>
19	DLOCK ²	<p>Data cache lock</p> <p>0 Normal operation</p> <p>1 All the ways of the data cache are locked. A locked cache supplies data normally on a read hit but is treated as a cache-inhibited transaction on a miss. On a miss, a load transaction still reads a full cache line from the L2 or MPX bus but does not reload that line into the L1. Any store miss is treated like a write-through store and the transaction occurs on the MPX bus with the $w\overline{t}$ signal asserted. A snoop hit to a locked L1 data cache operates as if the cache were not locked. A cache block invalidated by a snoop remains invalid until the cache is unlocked. Note that setting this bit has the same effect as setting LDSTCR[DCWL] to all ones. However, when this bit is set, LDSTCR[DCWL] is ignored. Refer to Chapter 3, “L1 and L2 Cache Operation,” for further details.</p> <p>To prevent locking during a cache access, a sync instruction must precede the setting of DLOCK and a sync must follow.</p>
20	ICFI ⁵	<p>Instruction cache flash invalidate</p> <p>0 The instruction cache is not invalidated. The bit is cleared when the invalidation operation begins (the next cycle after the write operation to the register). The instruction cache must be enabled for the invalidation to occur.</p> <p>1 An invalidate operation is issued that marks the state of each instruction cache block as invalid. Cache access is blocked during this time. Setting ICFI clears all the valid bits of the blocks and sets the PLRU bits to point to way L0 of each set. When the L1 flash invalidate bits are set through an mtspr operation, the hardware automatically clears these bits in the next cycle (provided that the corresponding cache enable bits are set in HID0).</p> <p>Note that in the MPC603e processors, the proper use of the ICFI and DCFI bits was to set and clear them in two consecutive mtspr operations. Software that already has this sequence of operations does not need to be changed to run on the e600 core.</p>
21	DCFI ²	<p>Data cache flash invalidate</p> <p>0 The data cache is not invalidated. The bit is cleared when the invalidation operation begins (the next cycle after the write operation to the register).</p> <p>1 An invalidate operation is issued that marks the state of each data cache block as invalid without writing back modified cache blocks to memory. Cache access is blocked during this time. MPX bus accesses to the cache are signaled as a miss during invalidate-all operations. Setting DCFI clears all the valid bits of the blocks and the PLRU bits to point to way L0 of each set. When the L1 flash invalidate bits are set through an mtspr operation, the hardware automatically clears these bits in the next cycle. Note that setting DCFI invalidates the data cache regardless of whether it is enabled.</p> <p>Note that in the MPC603e processors, the proper use of the ICFI and DCFI bits was to set them and clear them in two consecutive mtspr operations. Software that already has this sequence of operations does not need to be changed to run on the e600 core.</p>

Table 2-7. HID0 Field Descriptions (continued)

Bits	Name	Description
22	SPD ¹	Speculative data cache and instruction cache access disable 0 Speculative bus accesses to nonguarded space (G = 0) from both the instruction and data caches is enabled. 1 Speculative bus accesses to nonguarded space in both caches is disabled. Thus, setting this bit prevents L1 data cache misses from going to the core memory subsystem until the instruction that caused the miss is next to complete. The HID0[SPD] bit also prevents instruction cache misses from going to the core memory subsystem until there are no unresolved branches. For more information on this bit and its effect on re-ordering of loads and stores, see Section 3.3.3.5, “Enforcing Store Ordering with Respect to Loads.”
23	XBSEN	Extended BAT block size enable. 0 Disables IBATnU[XBL] and DBATnU[XBL] bits and clears these bits to zero. 1 Enables IBATnU[XBL] and DBATnU[XBL] bits. BATnU[15–18] become the 4 MSBs of the extended 15-bit BL field (BATnU[15–29]). This allows for extended BAT block sizes of 512 Mbytes, 1 Gbyte, 2 Gbytes, and 4 Gbytes. If HID0[XBSEN] is set at startup and then cleared after startup, the XBL bits will not clear but stay the same as they were set at startup. HID0[XBSEN] should be set once at startup and once set should not be cleared. When HID0[XBSEN] is set at startup, and then HID0[XBSEN] is cleared, the IBATnU[XBL] and DBATnU[XBL] bits are not cleared but stay the same as what was set at startup. If backwards compatibility with previous processors is a concern, then HID0[XBSEN] should stay cleared so that the XBL bits are treated as zeros. This allows the BAT translation to have a maximum block length of 256 Mbytes.
24	SGE ⁷	Store gathering enable 0 Store gathering is disabled. 1 Integer store gathering is performed as described in 3.1.2.3, “Store Gathering/Merging,” and Section 6.4.4.2, “Store Gathering.”
25	—	Reserved. Defined as DCFA on some earlier processors.
26	BTIC ¹	Branch target instruction cache enable. Used to enable use of the 128-entry branch instruction cache. 0 The BTIC contents are invalidated and the BTIC behaves as if it were empty. New entries cannot be added until the BTIC is enabled. 1 The BTIC is enabled and new entries can be added. The BTIC is flushed by context synchronization, which is required after a move to HID0. Thus, if the synchronization rules are followed, modifying this BTIC bit implicitly flushes the BTIC. See Chapter 6, “Instruction Timing,” for further details.
27	LRSTK ¹	Link register stack enable 0 Link register prediction is disabled. 1 Allows bclr and bclrI instructions to predict the branch target address using the link register stack which can accelerate returns from subroutines. See Chapter 6, “Instruction Timing,” for further details.
28	FOLD ¹	Branch folding enable 0 Branch folding is disabled. All branches are dispatched to the completion queue. 1 Branch folding is enabled, allowing branches to be folded out of the instruction prefetch stream before dispatch. The e600 core attempts to fold branches that do not modify the link and or count register. Note that a branch that is taken or predicted as taken is folded regardless of where it is in the IQ; however, a branch that is predicted as not taken must be dispatched (cannot be fall-through folded) if it is in one of the dispatch entries (IQ0–IQ2) the cycle after it is decoded. See Section 6.7.1.5, “Branch Folding,” and Section 6.7.2, “Dispatch Unit Resource Requirements,” for further details.

Table 2-7. HID0 Field Descriptions (continued)

Bits	Name	Description
29	BHT ¹	Branch history table enable 0 BHT disabled. The e600 core uses static branch prediction as defined by the PowerPC architecture (UISA) for those branch instructions the BHT would have otherwise used to predict (that is, those that use the CR or CTR mechanism to determine direction). For more information on static branch prediction, see “Conditional Branch Control,” in Chapter 4 of the <i>Programming Environments Manual</i> . 1 Allows the use of dynamic prediction in the 2048-entry branch history table (BHT). The BHT is disabled at power-on reset. All entries are set to weakly, not-taken.
30	NOPDST ²	No-op dst , dstt , dstst , and dststt instructions 0 The dst , dstt , dstst , and dststt instructions are enabled. 1 The dst , dstt , dstst , and dststt instructions are no-oped globally, and all previously executed dst streams are cancelled.
31	NOPTI ⁷	No-op the data cache touch instructions 0 The dcbt and dcbtst instructions are enabled. 1 The dcbt and dcbtst instructions are no-oped globally.

¹ A context synchronizing instruction must follow the **mtspr**.

² A **dssall** and **sync** must precede an **mtspr** and then a **sync** and context synchronizing instruction must follow. Note that if a user is not using the AltiVec data streaming instructions, then a **dssall** is not necessary prior to accessing the HID0[DCE] or HID0[DCFI] bit.

³ A context synchronizing instruction must precede an **mtspr** and a branch instruction should follow. The branch instruction may be either conditional or unconditional. It ensures that all subsequent branch instructions see the newly initialized BHT values. For correct results, the BHT should be disabled (HID0[BHT] = 0) before setting BHTCLR.

⁴ A **dssall** and **sync** must precede an **mtspr** and then a **sync** and a context-synchronizing instruction must follow. Alteration of HID0[XAEN] must be done with caches and translation disabled. The caches and TLBs must be flushed before they are re-enabled after the XAEN bit is altered. Note that if a user is not using the AltiVec data streaming instructions, then a **dssall** is not necessary prior to accessing the HID0[XAEN] bit.

⁵ A context synchronizing instruction must immediately follow an **mtspr**. An **mtspr** instruction for HID0 should not modify either of these bits at the same time it modifies another bit that requires additional synchronization.

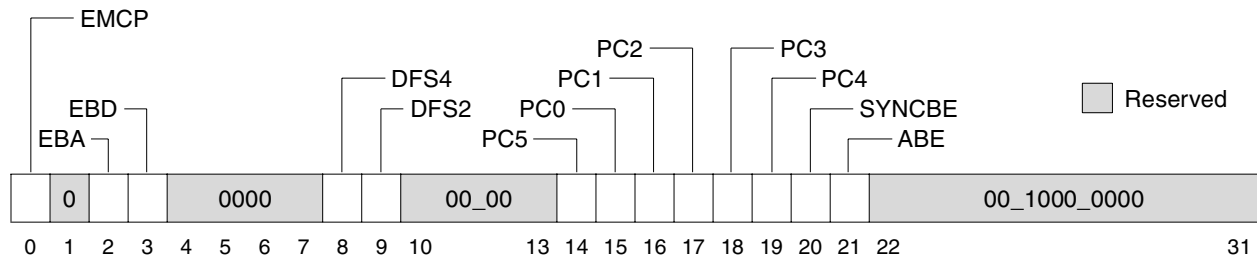
⁶ A context synchronizing instruction must precede and follow an **mtspr**.

⁷ An **mtspr** must follow a **sync** and a context synchronizing instruction.

HID0 can be accessed with **mtspr** and **mf spr** using SPR 1008. All **mtspr** instructions should be followed by a context synchronization instruction such as **isync**; for specific details see [Section 2.3.2.4, “Synchronization.”](#)

2.1.6.2 Hardware Implementation-Dependent Register 1 (HID1)

The hardware implementation-dependent register 1 (HID1) reflects the state of the *pll_cfg*[0:5] signals and controls other functions. At reset, PC bits 0–5 reflect the values on the *pll_cfg*[0:5] pins. The values of these bits are updated by the processor when DFS is enabled. The HID1 bits are shown in [Figure 2-7](#).


Figure 2-7. Hardware Implementation-Dependent Register 1 (HID1)

The HID1 bits are described in [Table 2-8](#).

Table 2-8. HID1 Field Descriptions

Bits ¹	Name	Description
0	EMCP	Machine check signal enable 0 Machine check is disabled. 1 Machine check input signal (\overline{mcp}) is enabled to cause machine check errors or checkstops.
1	—	Reserved
2	EBA	Enable/disable MPX bus address bus parity checking. 0 Address bus parity checking is disabled. 1 Allows an address bus parity error to cause a checkstop if MSR[ME] = 0 or a machine check interrupt if MSR[ME] = 1. Clearing EBA and EBD allows the processor to operate with memory subsystems that do not generate parity. The e600 core always generates parity regardless of whether checking is enabled or disabled.
3	EBD	Enable/disable MPX bus data parity checking. 0 Data parity checking is disabled. 1 Allows a data bus parity error to cause a checkstop if MSR[ME] = 0 or a machine check interrupt if MSR[ME] = 1. Clearing EBA and EBD allows the processor to operate with memory subsystems that do not generate parity. The e600 core always generates parity regardless of whether checking is enabled or disabled.
4–7	—	Reserved
8	DFS4	Dynamic frequency switching (DFS) divide-by-four mode. 0 DFS divide-by-four mode is disabled. 1 DFS divide-by-four mode is enabled. When both DFS2 and DFS4 bits are set, divide-by-four mode is selected. In the e600 core, dynamic frequency switching (DFS) divide-by-four mode can be controlled by hardware through the <i>dfs4</i> signal. The value of this field must match what the integrated device supports.
9	DFS2	Dynamic frequency switching (DFS) divide-by-two mode. 0 DFS divide-by-two mode is disabled. 1 DFS divide-by-two mode is enabled. Note that the divisors are only applicable to the core-to-MPX bus ratio chosen at reset by the <i>pll_cfg</i> pins. If the HID1 settings select a ratio that is not supported (see the specific device hardware specifications for supported ratios), the setting of the HID1[DFS2] bit is ignored. In the e600 core, dynamic frequency switching (DFS) divide-by-two mode can be controlled by hardware through the <i>dfs2</i> signal. The value of this field must match what the integrated device supports.
10–13	—	Reserved

Table 2-8. HID1 Field Descriptions (continued)

Bits ¹	Name	Description
14	PC5	PLL configuration bit 5 (read-only). Reflects the state of the PLL multiplier.
15	PC0	PLL configuration bit 0 (read-only). Reflects the state of the PLL multiplier.
16	PC1	PLL configuration bit 1 (read-only). Reflects the state of the PLL multiplier.
17	PC2	PLL configuration bit 2 (read-only). Reflects the state of the PLL multiplier.
18	PC3	PLL configuration bit 3 (read-only). Reflects the state of the PLL multiplier.
19	PC4	PLL configuration bit 4 (read-only). Reflects the state of the PLL multiplier.
20	SYNCBE	Address broadcast enable for sync , eieio 0 Address broadcasting of sync , and eieio is disabled. 1 Address broadcasting of sync , and eieio is enabled. Note this bit must be set in MP systems and systems that reorder stores.
21	ABE	Address broadcast enable for dcbf , dcbst , dcbi , icbi , tlbie , and tlbsync . 0 Address broadcasting of dcbf , dcbst , dcbi , icbi , tlbie , and tlbsync is disabled. Note that when HID1[ABE] is cleared this does not exclude all cache operations from the MPX bus—just icbi , tlbie , and tlbsync . 1 Address broadcasting for cache control operations (dcbf , dcbst , dcbi , icbi) and TLB control operations (tlbie and tlbsync) is enabled. Note that whether the broadcast occurs depends on the setting of the M bit of WIMG and whether the access causes a hit to modified memory. See Section 3.7.1, “MPX bus Operations Caused by Cache Control Instructions,” for more information on broadcast operations. The ABE bit must be set for MP systems.
22–31	—	Reserved. Read as 0b00_1000_0000.

¹ A sync and context synchronizing instruction must follow an `mtspr`.

NOTE

The required software sequence for setting or clearing the HID1[DFS2] bit is as follows:

```
sync
mtspr HID1
sync
isync
```

HID1 can be accessed with `mtspr` and `mfspr` using SPR 1009. All `mtspr` instructions should be followed by a `sync` and context synchronization instruction; for specific details, see [Section 2.3.2.4, “Synchronization.”](#)

2.1.6.3 Memory Subsystem Control Register (MSSCR0)

The memory subsystem control register (MSSCR0), shown in [Figure 2-8](#), is used to configure and operate the memory subsystem for the e600 core. It is accessed as SPR 1014. The MSSCR0 is initialized to all zeros except for the read-only bits.

Because MSSCR0 alters how the e600 responds to snoop requests, it is important that changes to the values of the fields in MSSCR0 are handled correctly.

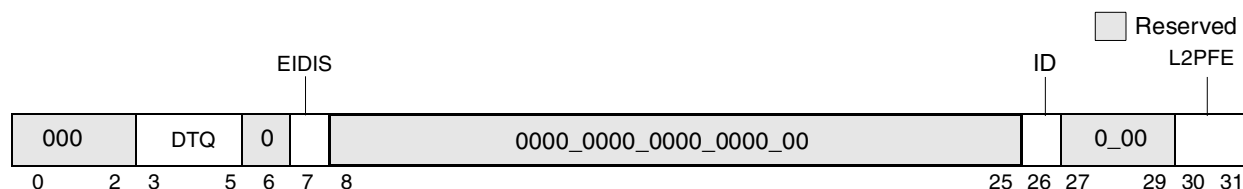


Figure 2-8. Memory Subsystem Control Register (MSSCR0)

Table 2-9 describes the MSSCR0 fields.

Table 2-9. MSSCR0 Field Descriptions

Bits	Name	Function
0–2	—	Reserved
3–5	DTQ	DTQ size. Determines the maximum number of outstanding data bus transactions that the e600 core can support. The DTQ bit values are as follows: 000 8 entries 001 16 entries 010 2 entries 011 3 entries 100 4 entries 101 5 entries 110 6 entries 111 7 entries The value of this field must match what the integrated device supports.
6	—	Reserved
7	EIDIS	Disable external intervention 0 Interventions external to the core occur. 1 The e600 core performs external pushes instead of external interventions. External interventions are disabled.
8–15	—	Reserved
16–17	BMODE	Bus mode (read-only). Indicates whether the core interface uses the 60x or MPX bus protocol as described in Chapter 8, “Core Interface.” 00 60x bus mode 01 Reserved 10 MPX bus mode 11 Reserved
18–25	—	Reserved. Normally cleared. Used in debug. Writing non-zero values may cause boundedly undefined results.
26	ID	Processor identification. Sets the processor ID to either processor 0 or 1. Determined by the integration logic. Software can then find processor 0 and use it to re-identify the other processors by writing unique values to the PIR of the other CPUs.

Table 2-9. MSSCR0 Field Descriptions (continued)

Bits	Name	Function
27–29	—	Reserved. Read as zeros.
30–31	L2PFE	L2 prefetching enabled. The following values determine the number of L2 prefetch engines enabled as follows: 00 L2 prefetching disabled, no prefetch engines 01 One prefetch engine enabled 10 Two prefetch engines enabled 11 Three prefetch engines enabled These bits enable alternate sector prefetching in the 2-sectored L2 cache; up to 3 outstanding prefetch engines may be active. See the integrated device reference manual for specific recommendations.

2.1.6.4 Memory Subsystem Status Register (MSSSR0)

The memory subsystem status register (MSSSR0), shown in [Figure 2-9](#), is used to report parity in the L2 cache and MSS enabled error status. It is accessed as SPR 1015. The MSSSR0 is initialized to all 0s except for the read-only bits.

Note that tag and data parity and data ECC errors are reported in the error detect (L2ERRDET) register whether or not error reporting is enabled. The corresponding bit in MSSSR0 is set only if error reporting is enabled.

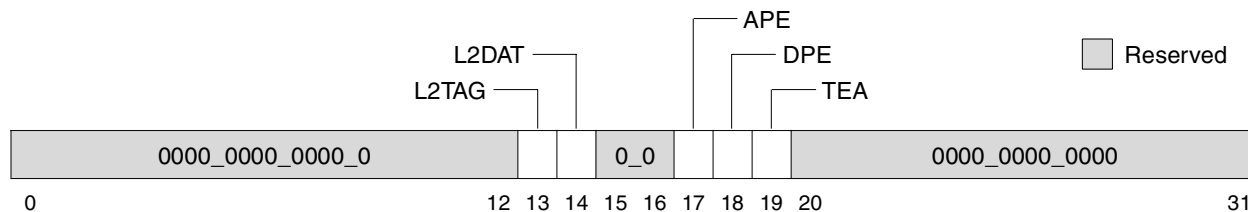


Figure 2-9. MSS Status Register (MSSSR0)

[Table 2-10](#) describes MSSSR0 fields.

Table 2-10. MSSSR0 Field Descriptions

Bits	Name	Description
0–12	—	Reserved. Normally cleared. Used in debug. Writing nonzero values may cause boundedly undefined results.
13	L2TAG	L2 tag parity error 0 L2 tag parity error not detected. 1 L2 tag parity error detected.
14	L2DAT	L2 data parity error 0 L2 data parity error not detected. 1 L2 data parity error detected.
15	—	Reserved. Normally cleared. Used in debug. Writing nonzero values may cause boundedly undefined results.

Table 2-10. MSSSR0 Field Descriptions (continued)

Bits	Name	Description
16	—	Reserved. Normally cleared. Used in debug. Writing nonzero values may cause boundedly undefined results.
17	APE	Address bus parity error 0 Address bus parity error not detected. 1 Address bus parity error detected.
18	DPE	Data bus parity error 0 Data bus parity error not detected. 1 Data bus parity error detected.
19	TEA	Bus transfer error acknowledge 0 \overline{tea} not detected as asserted. 1 \overline{tea} detected as asserted.
20–31	—	Reserved

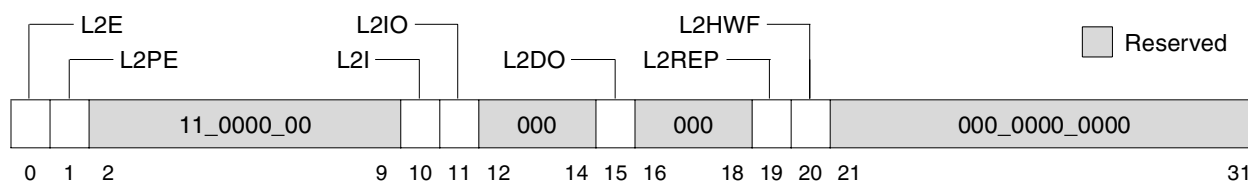
2.1.6.5 Instruction and Data Cache Registers

Several registers are used for configuring and controlling the various L1 and L2 caches. Along with the cache registers (L2CR, ICTRL, and LDSTCR), HID0 is used in configuring the caches. Details of how the various cache registers are used is discussed below. See [Chapter 3, “L1 and L2 Cache Operation,”](#) for further details on configuring the cache.

2.1.6.5.1 L2 Cache Control Register (L2CR)

The L2 cache control register (L2CR), shown in [Figure 2-10](#), is a supervisor-level, implementation-specific SPR used to configure and operate the L2 cache. It is cleared by a hard reset or power-on reset.

With the addition of ECC support, tag parity is controlled separately through the TPARDIS bit in the L2ERRDIS register. Data parity can only be enabled through L2CR[L2PE] if ECC is disabled in the L2ERRDIS register.


Figure 2-10. L2 Control Register (L2CR)

The L2 cache interface is described in [Chapter 3, “L1 and L2 Cache Operation.”](#) The bits of the L2CR are described in [Table 2-11](#).

Table 2-11. L2CR Field Descriptions

Bits	Name	Description
0	L2E	L2 enable. Used to enable the L2 cache. 0 The L2 cache is disabled and is not accessed for reads, snoops, or writes. 1 The L2 cache is enabled.
1	L2PE	L2 data parity checking enable 0 L2 data parity checking disabled 1 L2 data parity checking enabled if L2ERRDIS[MBECCDIS] = 1 and L2ERRDIS[SBECCDIS] = 1. If ECC is enabled (L2ERRDIS[MBECCDIS] = 0 or L2ERRDIS[SBECCDIS] = 0), setting L2PE has no effect; ECC checking will still be performed. Note: The L2ERRDIS register includes bits to enable/disable tag parity checking. Data parity can only be enabled with L2CR[L2PE] if ECC is disabled in the L2ERRDIS register. By default, tag parity and data ECC checking are enabled on the e600 core.
2–3	—	Reserved, will read as 0b11.
4–9	—	Reserved
10	L2I	L2 global invalidate 0 Do not perform global invalidate operation on the L2 cache. 1 L2 cache invalidate globally Invalidates the L2 cache globally by clearing the L2 status bits. This bit must not be set while the L2 cache is enabled. Note that L2I is automatically cleared when the global invalidate operation completes.
11	L2IO	L2 instruction-only. Causes the L2 cache to allocate lines for instruction cache transactions only. 0 The L2 cache allocates entries for data accesses that miss. 1 The L2 cache does not allocate entries for data accesses that miss in the L2. Data accesses that hit, instruction accesses, and system accesses are unaffected. If L2DO and L2IO are both set, no new lines are allocated in the L2 cache, effectively locking the entire cache.
12–14	—	Reserved
15	L2DO	L2 data only. L2 cache lines are allocated for data cache transactions only. 0 The L2 cache allocates entries for instruction accesses that miss. 1 The L2 cache does not allocate entries for instruction accesses that miss in the L2. Instruction accesses that hit in the L2, data accesses, and system accesses are unaffected. If both L2DO and L2IO are set, no new lines are allocated in the L2 cache, effectively locking the entire cache.
16–18	—	Reserved
19	L2REP	L2 replacement algorithm selection on a miss 0 Pseudo-random replacement algorithm is used (default) 1 3-bit counter replacement algorithm is used See Section 3.6.4.4, “L2 Cache Line Replacement Algorithms,” for more information.
20	L2HWF	L2 hardware flush 0 The L2 cache is not being flushed. 1 A 0->1 transition on this bit triggers a global flush of the entire L2 cache. All modified lines will be cast out to main memory. The cache must be locked by setting L2CR[L2DO] = 1 and L2CR[L2IO] = 1 before setting this bit. See Section 3.6.3.1.5, “Flushing of L1 and L2 Caches,” for more information.
21–23	—	Reserved
24–27	—	Reserved. Writing these bits will result in undefined L2 cache behavior.
28–31	—	Reserved

The L2CR register can be accessed with the **mtspr** and **mfspir** instructions using SPR 1017.

2.1.6.5.2 L2 Error Injection Mask High Register (L2ERRINJHI)

The L2 error injection mask high register (L2ERRINJHI), shown in [Figure 2-11](#), is a supervisor-level SPR used for error injection of the high word of the data path.



Figure 2-11. L2 Error Injection Mask High Register (L2ERRINJHI)

[Table 2-12](#) describes L2ERRINJHI[EIMASKHI].

Table 2-12. L2ERRINJHI Field Description

Bits	Name	Description
0–31	EIMASKHI	Error injection mask for the high word of the data path. A set bit corresponding to a data path bit causes that bit on the data path to be inverted on cache writes if data array error injection is enabled by setting L2ERRINJCTL[DERRIEN] = 1.

2.1.6.5.3 L2 Error Injection Mask High Register (L2ERRINJLO)

The L2 error injection mask low register (L2ERRINJLO), shown in [Figure 2-12](#), is a supervisor-level SPR used for error injection of the low word of the data path.

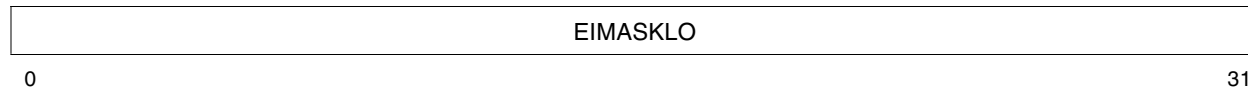


Figure 2-12. L2 Error Injection Mask Low Register (L2ERRINJLO)

[Table 2-13](#) describes L2ERRINJLO[EIMASKLO].

Table 2-13. L2ERRINJLO Field Description

Bits	Name	Description
0–31	EIMASKLO	Error injection mask for the low word of the data path. A set bit corresponding to a data path bit causes that bit on the data path to be inverted on cache writes if data array error injection is enabled by setting L2ERRINJCTL[DERRIEN] = 1.

2.1.6.5.4 L2 Error Injection Mask Control Register (L2ERRINJCTL)

The L2 error injection mask control register (L2ERRINJCTL), shown in [Figure 2-13](#), is a supervisor-level SPR used to configure error injection.

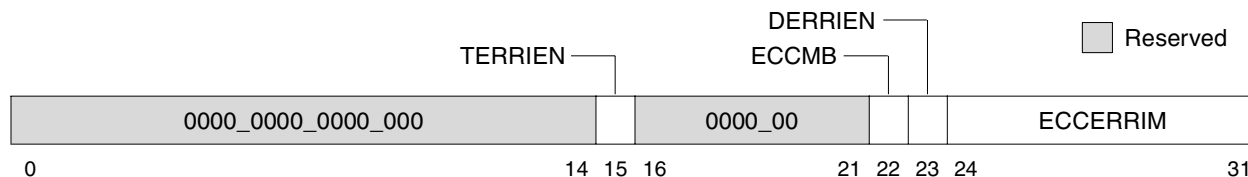


Figure 2-13. L2 Error Injection Mask Control Register (L2ERRINJCTL)

Table 2-14 describes L2ERRINJCTL fields.

Table 2-14. L2ERRINJCTL Field Descriptions

Bits	Name	Description
0–14	—	Reserved
15	TERRIEN	L2 tag array error injection enable 0 No tag errors are injected. 1 All subsequent entries written to the L2 tag array have the parity bit inverted.
16–21	—	Reserved
22	ECCMB	ECC mirror byte enable 0 ECC byte mirroring is disabled. 1 The most significant data path byte is mirrored onto the ECC byte if DERRIEN = 1.
23	DERRIEN	L2 data array error injection enable 0 No data errors are injected. 1 All subsequent entries written to the L2 data array have data or ECC bits inverted as specified in the data error injection masks and ECC error injection mask and/or data path byte mirrored onto ECC as specified by the ECC mirror byte enable bit, ECCMB. Note: if both ECC mirror byte and data error injection are enabled, ECC mask error injection is performed on the mirrored ECC.
24–31	ECCERRIM	Error injection mask for the ECC bits. A set bit corresponding to an ECC bit causes that bit to be inverted on cache writes if DERRIEN = 1.

2.1.6.5.5 L2 Error Capture Data High Register (L2CAPTDATAHI)

The L2 error capture data high register (L2CAPTDATAHI), shown in Figure 2-15, holds the high word of the L2 data that contains the detected error.

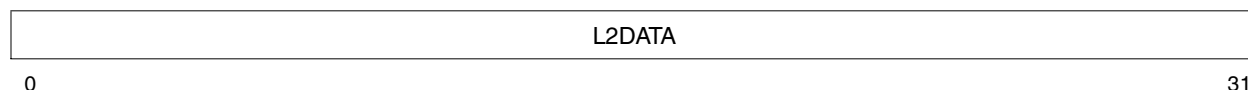


Figure 2-14. L2 Error Capture Data High Register (L2CAPTDATAHI)

Table 2-15 describes L2CAPTDATAHI[L2DATA].

Table 2-15. L2CAPTDATAHI Field Description

Bits	Name	Description
0–31	L2DATA	L2 data high word (read only)

2.1.6.5.6 L2 Error Capture Data Low Register (L2CAPTDATALO)

The L2 error capture data low register (L2CAPTDATALO), shown in [Figure 2-15](#), holds the low word of the L2 data that contains the detected error.

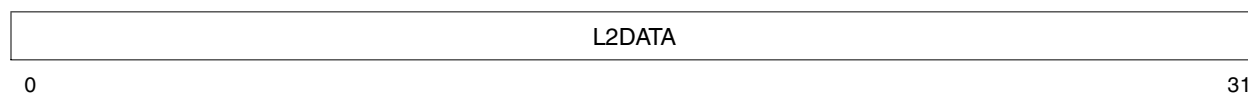


Figure 2-15. L2 Error Capture Data Low Register (L2CAPTDATALO)

[Table 2-16](#) describes L2CAPTDATALO[L2DATA].

Table 2-16. L2CAPTDATALO Field Description

Bits	Name	Description
0–31	L2DATA	L2 data low word (read only)

2.1.6.5.7 L2 Error Syndrome Register (L2CAPTECC)

The L2 error syndrome register (L2CAPTECC), shown in [Figure 2-16](#), is a supervisor-level SPR that contains the ECC syndrome and datapath ECC of the failing double word.

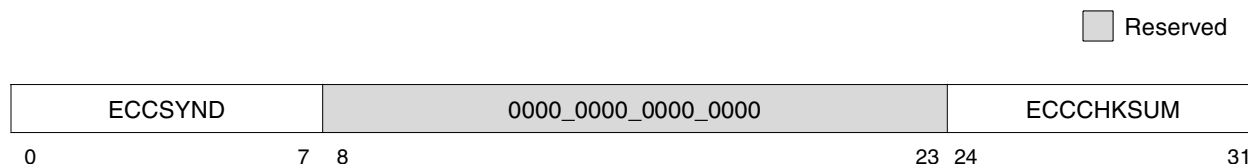


Figure 2-16. L2 Error Syndrome Register (L2CAPTECC)

[Table 2-17](#) describes L2CAPTECC fields.

Table 2-17. L2CAPTECC Field Descriptions

Bits	Name	Description
0–7	ECCSYND	The calculated ECC syndrome of the failing double word (read only)
8–23	—	Reserved
24–31	ECCCHKSUM	The datapath ECC of the failing double word (read only)

2.1.6.5.8 L2 Error Detect Register (L2ERRDET)

The L2 error detect register (L2ERRDET), shown in [Figure 2-17](#), is a supervisor-level SPR that shows the errors detected.

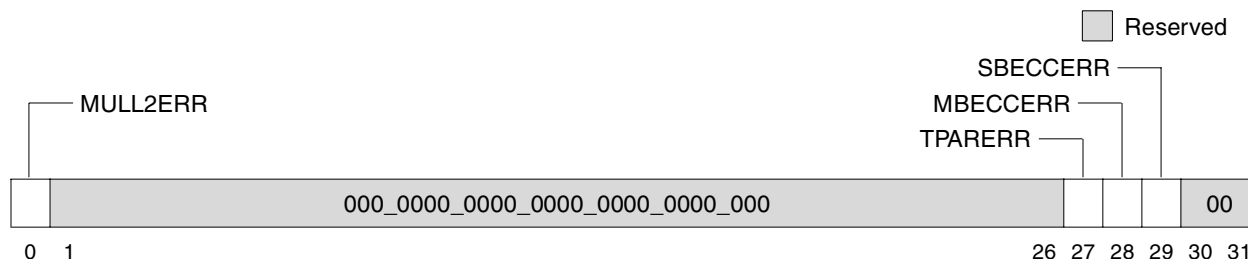


Figure 2-17. L2 Error Detect Register (L2ERRDET)

[Table 2-18](#) describes L2ERRDET fields.

Table 2-18. L2ERRDET Field Descriptions

Bits	Name	Description
0	MULL2ERR	Multiple L2 errors (Bit reset, write-1-to-clear) 0 Multiple L2 errors of the same type were not detected 1 Multiple L2 errors of the same type were detected Note that setting this bit to 1 clears it to a value of 0.
1–26	—	Reserved
27	TPARERR	Tag parity error (Bit reset, write-1-to-clear) 0 Tag parity error was not detected 1 Tag parity error was detected Note that setting this bit to 1 clears it to a value of 0.
28	MBECCERR	Multiple-bit ECC error (Bit reset, write-1-to-clear) 0 Multiple-bit ECC errors were not detected 1 Multiple-bit ECC errors were detected Note that setting this bit to 1 clears it to a value of 0.
29	SBECCERR	Single-bit ECC error (Bit reset, write-1-to-clear) 0 Single-bit ECC error was not detected 1 Single-bit ECC error was detected Note that setting this bit to 1 clears it to a value of 0.
30–31	—	Reserved

2.1.6.5.9 L2 Error Disable Register (L2ERRDIS)

The L2 error disable register (L2ERRDIS), shown in [Figure 2-18](#), is a supervisor-level SPR that disables and enables error detection. Note that the L2 cache must be disabled and flushed before enabling or disabling ECC to ensure that no errors occur. See [Section 3.6.3.4.1, “Enabling or Disabling ECC,”](#) and [Table 2-37](#) for the synchronization requirements required to enable or disable ECC.

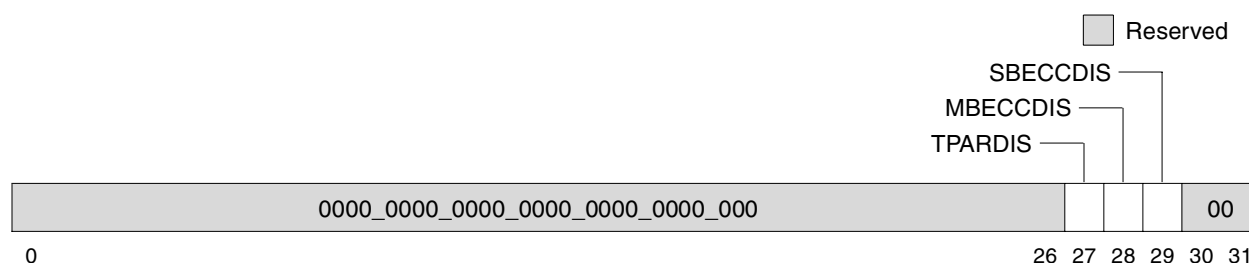


Figure 2-18. L2 Error Disable Register (L2ERRDIS)

[Table 2-19](#) describes L2ERRDIS fields.

Table 2-19. L2ERRDIS Field Descriptions

Bits	Name	Description
0–26	—	Reserved
27	TPARDIS	Tag parity error disable 0 Tag parity error detection enabled 1 Tag parity error detection disabled
28	MBECCDIS	Multiple-bit ECC error disable 0 Multiple-bit ECC error detection enabled 1 Multiple-bit ECC error detection disabled
29	SBECCDIS	Single-bit ECC error disable 0 Single-bit ECC error detection enabled 1 Single-bit ECC error detection disabled
30–31	—	Reserved

2.1.6.5.10 L2 Error Interrupt Enable Register (L2ERRINTEN)

The L2 error interrupt enable register (L2ERRINTEN), shown in Figure 2-19, is a supervisor-level SPR used to enable L2 error interrupts. When any of these error conditions exist and the corresponding bit in the L2ERRINTEN register is enabled, a machine check interrupt is generated.

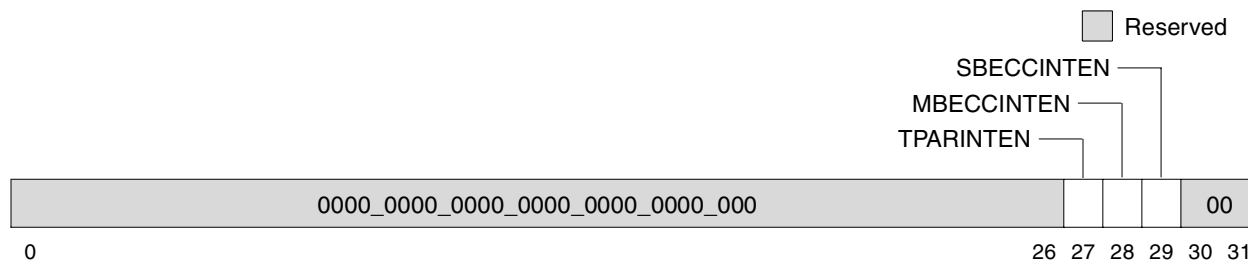


Figure 2-19. L2 Error Interrupt Enable Register (L2ERRINTEN)

Table 2-20 describes L2ERRINTEN fields.

Table 2-20. L2ERRINTEN Field Descriptions

Bits	Name	Description
0–26	—	Reserved
27	TPARINTEN	Tag parity error reporting enable 0 Tag parity error reporting disabled 1 Tag parity error reporting enabled.
28	MBECCINTEN	Multiple-bit ECC error reporting enable 0 Multiple-bit ECC error reporting disabled 1 Multiple-bit ECC error reporting enabled
29	SBECCINTEN	Single-bit ECC error reporting enable 0 Single-bit ECC error reporting disabled 1 Single-bit ECC error reporting enabled
30–31	—	Reserved

2.1.6.5.11 L2 Error Attributes Capture Register (L2ERRATTR)

The L2 error attributes capture register (L2ERRATTR), shown in Figure 2-20, is a supervisor-level SPR that describes the L2 error attributes. All the fields of the L2ERRATTR are read-only except for bit 31, VALINFO.

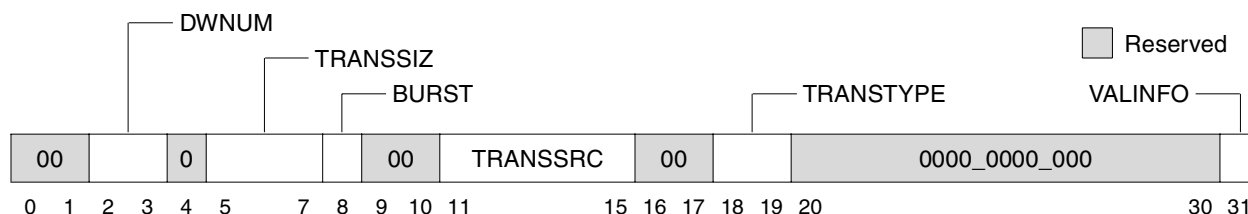


Figure 2-20. L2 Error Attributes Capture Register (L2ERRATTR)

Table 2-21 describes L2ERRATTR fields.

Table 2-21. L2ERRATTR Field Descriptions

Bits	Name	Description
0–1	—	Reserved
2–3	DWNUM	Double-word number of the detected error (data ECC errors only). Read only.
4	—	Reserved
5–7	TRANSSIZ	Transaction size for detected error (read only). Reserved for read transactions (TRANSTYPE = 10). The transaction size for a read to the L2 will always be a 32-byte burst. 000 8 bytes (single-beat) or reserved (burst) 001 1 byte (single-beat) or 16 bytes (burst) 010 2 bytes (single-beat) or 32 bytes (burst) 011 3 bytes (single beat) or reserved (burst) 100 4 bytes (single-beat) or reserved (burst) 101 5 bytes (single-beat) or reserved (burst) 110 6 bytes (single-beat) or reserved (burst) 111 7 bytes (single-beat) or reserved (burst)
8	BURST	Burst transaction for detected error. Read only. 0 Single-beat (≤ 64 bits) transaction 1 Burst transaction
9–10	—	Reserved
11–15	TRANSSRC	Transaction source for detected error. Read only. 00000 External (logic external to the core) 10000 Core (instruction) 10001 Core (data)
16–17	—	Reserved
18–19	TRANSTYPE	Transaction type for detected error. Read only. 00 Snoop (tag/status read) 01 Write 10 Read 11 Reserved
20–30	—	Reserved
31	VALINFO	L2 capture registers valid 0 L2 capture registers contain no valid information or no enabled errors were detected. 1 L2 capture registers contain information of the first detected error that has reporting enabled. Software must clear this bit to unfreeze error capture so error detection hardware can overwrite the capture address/data/attributes for a newly detected error.

2.1.6.5.12 L2 Error Address Error Capture Register (L2ERRADDR)

The L2 error address error capture register (L2ERRADDR), shown in Figure 2-21, is a supervisor-level SPR that shows the L2 address corresponding to bits 4–35 of the detected error.



Figure 2-21. L2 Error Address Error Capture Register (L2ERRADDR)

Table 2-22 describes L2ERRADDR.

Table 2-22. L2ERRADDR Field Description

Bits	Name	Description
0–31	L2ADDR	L2 address[4:35] corresponding to detected error (read only)

2.1.6.5.13 L2 Error Address Error Capture Register (L2ERREADDR)

The L2 error address error capture register (L2ERREADDR), shown in Figure 2-22, is a supervisor-level SPR that shows the L2 address corresponding to bits 0–3 of the detected error.

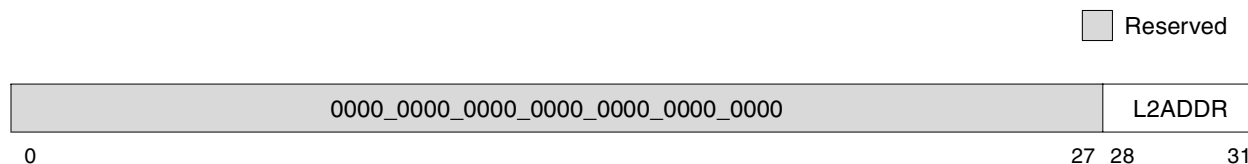


Figure 2-22. L2 Error Address Error Capture Register (L2ERREADDR)

Table 2-23 describes L2ERREADDR.

Table 2-23. L2ERREADDR Field Description

Bits	Name	Description
0–27	—	Reserved
28–31	L2EADDR	L2 address[0:3] corresponding to detected error (read only)

2.1.6.5.14 L2 Error Control Register (L2ERRCTL)

The L2 error control register (L2ERRCTL), shown in Figure 2-23, is a supervisor-level SPR that configures the L2 cache ECC error threshold and provides an L2 error count.

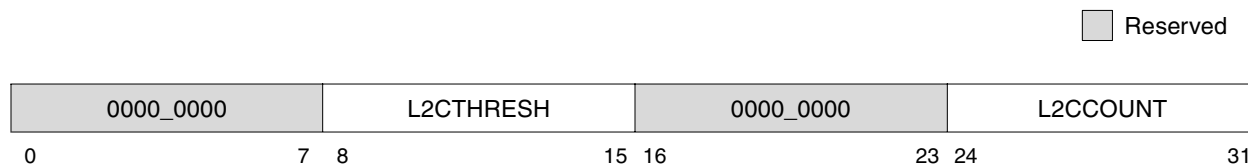


Figure 2-23. L2 Error Control Register (L2ERRCTL)

Table 2-25. ICTRL Field Descriptions (continued)

Bits	Name	Description
5	EDCE ²	Data cache parity error enable 0 When the bit is cleared, any parity error in the L1 data cache is masked and does not cause machine checks or checkstop 1 Enables data cache parity error reporting. When a data cache parity error occurs, a machine check interrupt is taken if MSR[ME] = 1. When this condition occurs, SRR1[2] is set. For details on the machine check interrupt see Section 4.6.2, “Machine Check Interrupt (0x00200).”
6–8	—	Reserved. Normally cleared. Used in debug. Writing nonzero values may cause boundedly undefined results.
9–22	—	Reserved. Read as zeros and ignores writes.
23	EICP	Enable instruction cache parity checking 0 Instruction cache parity disabled 1 When the EICP bit is set, the parity of any instructions fetched from the L1 instruction cache is checked. Any errors found are reported as instruction cache parity errors in SRR1. If EICE is also set, these instruction cache errors cause a machine check or checkstop. If either EICP or EICE is cleared, instruction cache parity is ignored. Note that when parity checking and error reporting are both enabled, errors are reported even on speculative fetches that are never actually executed. Correct instruction cache parity is always loaded into the L1 instruction cache regardless of whether checking is enabled or not.
24–31	ICWL ¹	Instruction cache way lock 0 Instruction cache way lock disabled. 1 Instruction cache way lock enabled. Each bit in ICWL corresponds to a way of the L1 instruction cache. Bit 24 corresponds to way 0, and bit 31 corresponds to way 7. Setting a bit locks the corresponding way in the instruction cache. Setting all 8 bits of ICWL is equivalent to locking the entire instruction cache. When all 8 ICWL bits are set, the e600 core behaves the same as when HID0[ILOCK] is set. See Section 2.1.6.1, “Hardware Implementation-Dependent Register 0 (HID0).” for details. See Chapter 3, “L1 and L2 Cache Operation,” for suggestions on how to keep the PLRU replacement algorithm symmetrical, and for synchronization requirements for modifying ICWL.

¹ A context synchronizing instruction must precede and follow an mtspr.

² A dssall and sync must precede an mtspr and then a sync and context synchronizing instruction must follow. Note that if a user is not using the Altivec data streaming instructions, then a dssall is not necessary prior to accessing the ICTRL[EDCE] bit.

ICTRL can be accessed with the **mtspr** and **mfspir** instructions using SPR 1011.

2.1.6.5.16 Load/Store Control Register (LDSTCR)

The load/store control register (LDSTCR) provides a way to lock the ways for the L1 data cache. The LDSTCR is shown in [Figure 2-29](#).

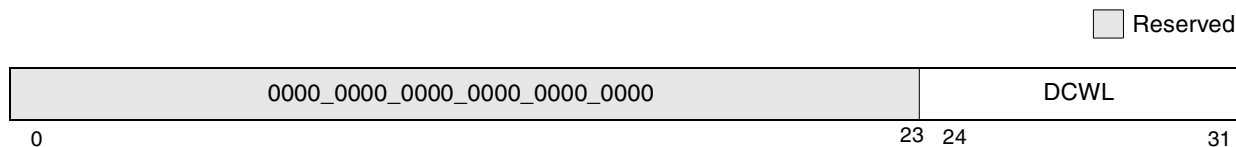


Figure 2-25. Load/Store Control Register (LDSTCR)

Table 2-30 describes the bit fields for the LDSTCR register.

Table 2-26. LDSTCR Field Descriptions

Bits	Name	Description
0–23	—	Reserved. Writing nonzero values may cause boundedly undefined results.
24–31	DCWL	Data cache way lock. Each bit in DCWL corresponds to a way of the L1 data cache. Bit 24 corresponds to way 0, and bit 31 corresponds to way 7. 0 Each cleared bit corresponds to the corresponding way not being locked in the L1 data cache. 1 Each set bit locks the corresponding way in the L1 data cache. When DCWL[24–31] are all set, it is equivalent to locking the entire L1 data cache and the e600 core behaves the same as if HID0[DLOCK] is set. “Chapter 3, “L1 and L2 Cache Operation,” describes how to keep the PLRU replacement algorithm symmetrical and for more information on synchronization requirements with LDSTCR.

The LDSTCR register can be accessed with the **mtspr** and **mfspir** instructions using SPR 1016. For synchronization requirements on the register see [Section 2.3.2.4, “Synchronization.”](#)

2.1.6.6 Instruction Address Breakpoint Register (IABR)

The instruction address breakpoint register (IABR), shown in [Table 2-26](#), supports the instruction address breakpoint interrupt. When this interrupt is enabled, instruction fetch addresses are compared with an effective address stored in the IABR. If the word specified in the IABR is fetched, the instruction breakpoint handler is invoked. The instruction that triggers the breakpoint does not execute before the handler is invoked. For more information, see [Section 4.6.16, “Instruction Address Breakpoint Interrupt \(0x01300\).”](#) The IABR can be accessed with **mtspr** and **mfspir** using SPR 1010. The e600 core requires that an **mtspr**[IABR] be followed by a context synchronizing instruction. The e600 core may not generate a breakpoint response for that context synchronizing instruction if the breakpoint was enabled by **mtspr**[IABR] immediately preceding it. The e600 core cannot block a breakpoint response on the context synchronizing instruction if the breakpoint was disabled by **mtspr**[IABR] immediately preceding it. For more information on synchronization see [Section 2.3.2.4.1, “Context Synchronization.”](#)

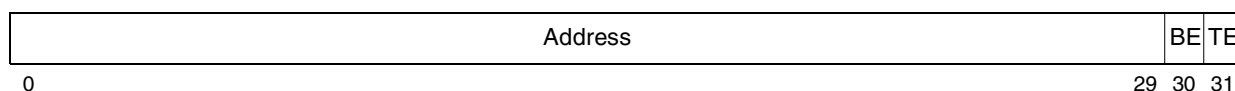


Figure 2-26. Instruction Address Breakpoint Register (IABR)

The IABR bits are described in [Table 2-27](#).

Table 2-27. IABR Field Descriptions

Bits ¹	Name	Description
0–29	Address	Word instruction breakpoint address to be compared with EA[0–29] of the next instruction
30	BE	Breakpoint enabled. Setting this bit enables breakpoint address checking.
31	TE	Translation enable IABR[TE] must equal MSR[IR] in order for a match to be signaled. When IABR[TE] and MSR[IR] = 0 or when IABR[TE] and MSR[IR] = 1, a match is signaled.

¹ A context synchronizing instruction must follow an **mtspr**.

2.1.6.7 Memory Management Registers Used for Software Table Searching

This section describes the registers used by the e600 core when software table searching is enabled (HID0[STEN] = 1) and a TLB miss interrupt occurs. Software table searching is described in detail in Chapter 5, “Memory Management.”

2.1.6.7.1 TLB Miss Register (TLBMISS)

The TLBMISS register is automatically loaded by the e600 core when software table searching is enabled (HID0[STEN] = 1) and a TLB miss interrupt occurs. Its contents are used by the TLB miss interrupt handlers (the software table search routines) to start the search process. Note that the e600 core always loads a big-endian address into the TLBMISS register. This register is read-only. The TLBMISS register has the format shown in Figure 2-27 for the e600 core.



Figure 2-27. TLBMISS Register

Table 2-28 describes the bits in the TLBMISS register.

Table 2-28. TLBMISS Register—Field and Bit Descriptions

Bits	Name	Description
0–30	PAGE	Effective page address. Stores EA[0–30] of the access that caused the TLB miss interrupt.
31	LRU	Least recently used way of the addressed TLB set. The LRU bit can be loaded into bit 31 of rB, prior to execution of tlbli or tlbld to select the way to be replaced for a TLB miss. However, this value should be inverted in rB prior to execution of tlbli or tlbld for a TLB miss interrupt caused by the need to update the C-bit.

TLBMISS can be accessed with **mtspr** and **mfspr** using SPR 980.

2.1.6.7.2 Page Table Entry Registers (PTEHI and PTELO)

The PTEHI and PTELO registers are used by the **tlbld** and **tlbli** instructions to create a TLB entry. When software table searching is enabled (HID0[STEN] = 1) and a TLB miss interrupt occurs, the bits of the page table entry (PTE) for this access are located by software and saved in the PTE registers. Figure 2-28 shows the format for two supervisor registers, PTEHI and PTELO, respectively.

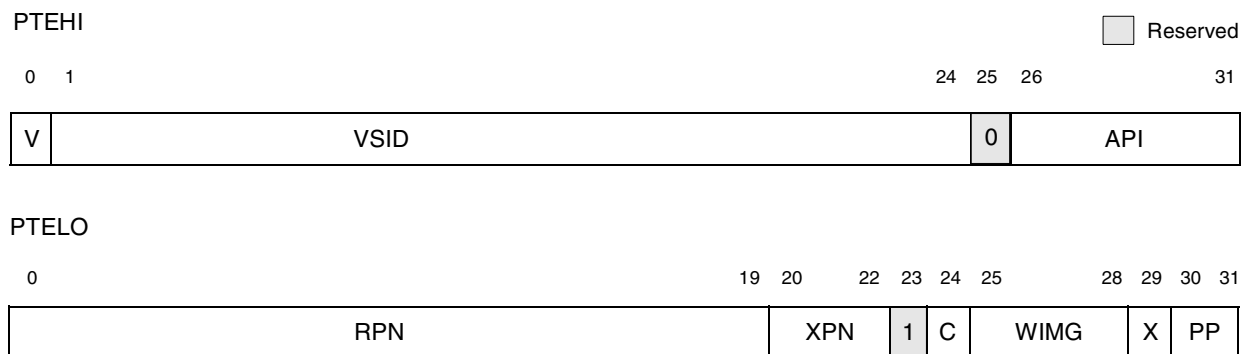


Figure 2-28. PTEHI and PTELO Registers—Extended Addressing

Note that the contents of PTEHI are automatically loaded when any of the three software table search interrupts is taken. PTELO is loaded by the software table search routines (the TLB miss interrupt handlers) based on the valid PTE located in the page tables prior to execution of the **tlbli** or **tlbld** instruction.

Table 2-29 lists the corresponding bit definitions for the PTEHI and PTELO registers.

Table 2-29. PTEHI and PTELO Bit Definitions

Register	Bit	Name	Description
PTEHI	0	V	Entry valid (V = 1) or invalid (V = 0). Always set by the processor on a TLB miss interrupt.
	1–24	VSID	Virtual segment ID. The corresponding SR[VSID] field is copied to this field.
	25	—	Reserved. Corresponds to the hash function identifier in PTE.
	26–31	API	Abbreviated page index. TLB miss interrupts will set this field with bits from TLBMISS[4–9] which are bits from the effective address for the access that caused the software table search operation. The tlbld and tlbli instructions ignore the API bits in PTEHI register and get the API from the instruction's operand, rB . However, for future compatibility, the API in rB should match the PTEHI[API].
PTELO	0–19	RPN	Physical page number
	20–22	XPN	Extended page number The XPN field provides the physical address bits, PA[0–2].
	23	—	Reserved. Corresponds to the reference bit in a PTE. The referenced bit is not stored in the page tables, so this bit is ignored in the PTELO register. To simplify software, this bit is hard-wired to 1. All the other bits in PTELO correspond to the bits in the low word of the PTE.
	24	C	Changed bit
	25–28	WIMG	Memory/cache control bits
	29	X	Extended page number The X field provides the physical address bit 3, PA[3].
	30–31	PP	Page protection bits

When extended addressing is not enabled, ($HID0[XAEN] = 0$), the software must clear the PTELO[XPN] and PTELO[X] bits; otherwise whatever values are in the fields become the four most-significant bits of

the physical address. **Note:** The PTEHI register is accessed with **mtspr** and **mfspir** as SPR 981 and PTELO is accessed as SPR 982.

2.1.6.8 Thermal Management Register

The e600 core provides an instruction cache throttling mechanism to reduce the instruction execution rate without the complexity and overhead of dynamic clock control. When used with dynamic power management, instruction cache throttling provides the system designer with a flexible way to control device temperature while allowing the processor to continue operating. Note that performance does degrade when instruction cache throttling is enabled to reduce junction temperature; entering short bursts of nap or sleep is a superior method for reducing thermal output and power consumption.

2.1.6.8.1 Instruction Cache Throttling Control Register (ICTC)

Reducing the rate of instruction fetching can control junction temperature without the complexity and overhead of dynamic clock control. System software can control instruction forwarding by writing a nonzero value to the ICTC register, a supervisor-level register shown in [Figure 2-29](#). The overall junction temperature reduction comes from the dynamic power management of each functional unit when the e600 core is idle in between instruction fetches.

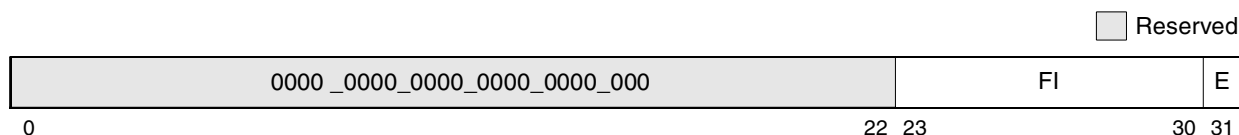


Figure 2-29. Instruction Cache Throttling Control Register (ICTC)

[Table 2-30](#) describes the bit fields for the ICTC register.

Table 2-30. ICTC Field Descriptions

Bits	Name	Description
0–22	—	Reserved. The bits should be cleared.
23–30	INTERVAL	Instruction forwarding interval expressed in core clocks. When throttling is enabled, the interval field specifies the minimum number of cycles between instructions being dispatched. (The core dispatches one instruction every INTERVAL cycle.) The minimum interval for throttling control is 2 cycles. 0x00, 0x01, 0x02 One instruction dispatches every 2 core clocks. 0x03 One instruction dispatches every 3 core clocks ... 0xFF One instruction dispatches every 255 core clocks.
31	E	Enable instruction throttling 0 Instructions dispatch normally. 1 Only one instruction dispatches every INTERVAL cycles.

Instruction cache throttling is enabled by setting ICTC[E] and writing the instruction forwarding interval into ICTC[INTERVAL]. A context synchronizing instruction should be executed after a move to the ICTC register to ensure that it has taken effect. Enabling, disabling, or changing the instruction forwarding interval affects instruction forwarding immediately.

The ICTC register can be accessed with the **mtspr** and **mfspir** instructions using SPR 1019.

2.1.6.9 Performance Monitor Registers

This section describes the registers used by the performance monitor, which is described in [Chapter 10](#), “Performance Monitor.”

2.1.6.9.1 Monitor Mode Control Register 0 (MMCR0)

The monitor mode control register 0 (MMCR0), shown in [Figure 2-30](#), is a 32-bit SPR provided to specify events to be counted and recorded. If the state of MSR[PR] and MSR[PMM] matches a state specified in MMCR0, then counting is enabled see [Section 10.4](#), “Event Counting,” for further details. The MMCR0 can be accessed only in supervisor mode. User-level software can read the contents of MMCR0 by issuing an `mfsprr` instruction to UMMCR0, described in [Section 2.1.6.9.2](#), “User Monitor Mode Control Register 0 (UMMCR0).”

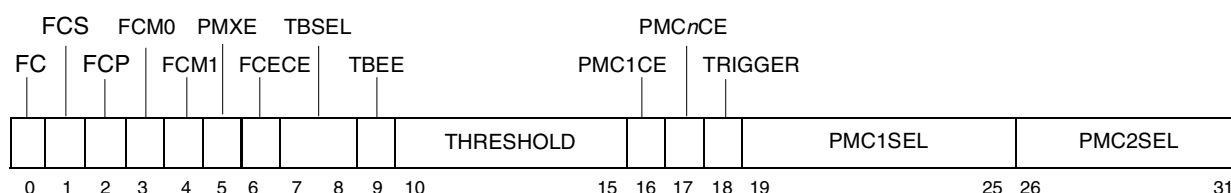


Figure 2-30. Monitor Mode Control Register 0 (MMCR0)

This register is automatically cleared at power-up. Reading this register does not change its contents. [Table 2-31](#) describes MMCR0 fields.

Table 2-31. MMCR0 Field Descriptions

Bits	Name	Description
0	FC	Freeze counters 0 The PMCs are incremented (if permitted by other MMCR bits). 1 The PMCs are not incremented (performance monitor counting is disabled). The processor sets this bit when an enabled condition or event occurs and MMCR0[FCECE] = 1. Note that SIAR is not updated if performance monitor counting is disabled.
1	FCS	Freeze counters in supervisor mode 0 The PMCs are incremented (if permitted by other MMCR bits). 1 The PMCs are not incremented if MSR[PR] = 0.
2	FCP	Freeze counters in user mode 0 The PMCs are incremented (if permitted by other MMCR bits). 1 The PMCs are not incremented if MSR[PR] = 1.
3	FCM1	Freeze counters while mark = 1 0 The PMCs are incremented (if permitted by other MMCR bits). 1 The PMCs are not incremented if MSR[PMM] = 1.
4	FCM0	Freeze counters while mark = 0 0 The PMCs are incremented (if permitted by other MMCR bits). 1 The PMCs are not incremented if MSR[PMM] = 0.

Table 2-31. MMCR0 Field Descriptions (continued)

Bits	Name	Description
5	PMXE	<p>Performance monitor interrupt enable</p> <p>0 Performance monitor interrupts are disabled.</p> <p>1 Performance monitor interrupts are enabled until a performance monitor interrupt occurs, at which time MMCR0[PMXE] is cleared.</p> <p>Software can clear PMXE to prevent performance monitor interrupts. Software can also set PMXE and then poll it to determine whether an enabled condition or event occurred.</p>
6	FCECE	<p>Freeze counters on enabled condition or event</p> <p>0 The PMCs are incremented (if permitted by other MMCR bits).</p> <p>1 The PMCs are incremented (if permitted by other MMCR bits) until an enabled condition or event occurs when MMCR0[TRIGGER] = 0, at which time MMCR0[FC] is set. If the enabled condition or event occurs when MMCR0[TRIGGER] = 1, FCECE is treated as if it were 0.</p> <p>The use of the trigger and freeze counter conditions depends on the enabled conditions and events described in Section 10.2, “Performance Monitor Interrupt.”</p>
7–8	TBSEL	<p>Time base selector. Selects the time base bit that can cause a time base transition event (the event occurs when the selected bit changes from 0 to 1).</p> <p>00 TBL[31]</p> <p>01 TBL[23]</p> <p>10 TBL[19]</p> <p>11 TBL[15]</p> <p>Time base transition events can be used to periodically collect information about processor activity. In multiprocessor systems in which the TB registers are synchronized among processors, time base transition events can be used to correlate the performance monitor data obtained by the several processors. For this use, software must specify the same TBSEL value for all the processors in the system. Because the time-base frequency is implementation-dependent, software should invoke a system service program to obtain the frequency before choosing a value for TBSEL.</p>
9	TBEE	<p>Time base event enable</p> <p>0 Time-base transition events are disabled.</p> <p>1 Time-base transition events are enabled. A time-base transition is signaled to the performance monitor if the TB bit specified in MMCR0[TBSEL] changes from 0 to 1. Time-base transition events can be used to freeze the counters (MMCR0[FCECE]), trigger the counters (MMCR0[TRIGGER]), or signal an interrupt (MMCR0[PMXE]).</p> <p>Changing the bits specified in MMCR0[TBSEL] while MMCR0[TBEE] is enabled may cause a false 0 to 1 transition that signals the specified action (freeze, trigger, or interrupt) to occur immediately.</p>
10–15	THRESHOLD	<p>Threshold. Contains a threshold value between 0 to 63. Two types of thresholds can be counted. The first type counts any event that lasts longer than the threshold value and uses MMCR2[THRESHMULT] to scale the threshold value by 2 or 32.</p> <p>The second type counts only the events that exceed the threshold value. This type does not use MMCR2[THRESHMULT] to scale the threshold value.</p> <p>By varying the threshold value, software can obtain a profile of the characteristics of the events subject to the threshold. For example, if PMC1 counts cache misses for which the duration exceeds the threshold value, software can obtain the distribution of cache miss durations for a given program by monitoring the program repeatedly using a different threshold value each time.</p>
16	PMC1CE	<p>PMC1 condition enable. Controls whether counter negative conditions due to a negative value in PMC1 are enabled.</p> <p>0 Counter negative conditions for PMC1 are disabled.</p> <p>1 Counter negative conditions for PMC1 are enabled. Such events can be used to freeze counters (MMCR0[FCECE]), trigger the counters (MMCR0[TRIGGER]), or signal an interrupt (MMCR0[PMXE]).</p>

Table 2-31. MMCR0 Field Descriptions (continued)

Bits	Name	Description
17	PMC n CE	PMC n condition enable. Controls whether counter negative conditions due to a negative value in any PMC n (that is, in any PMC except PMC1) are enabled. 0 Counter negative conditions for all PMC n s are disabled. 1 Counter negative conditions for all PMC n s are enabled. These events can be used to freeze the counters (MMCR0[FCECE]), trigger the counters (MMCR0[TRIGGER]), or signal an interrupt (MMCR0[PMXE]).
18	TRIGGER	Trigger 0 The PMCs are incremented (if permitted by other MMCR bits). 1 PMC1 is incremented (if permitted by other MMCR bits). The PMC n s are not incremented until PMC1 is negative or an enabled timebase or event occurs, at which time the PMC n s resume incrementing (if permitted by other MMCR bits) and MMCR0[TRIGGER] is cleared. The description of FCECE explains the interaction between TRIGGER and FCECE. Uses of TRIGGER include the following: <ul style="list-style-type: none"> Resume counting in the PMCns when PMC1 becomes negative without causing a performance monitor interrupt. Then freeze all PMCs (and optionally cause a performance monitor interrupt) when a PMCn becomes negative. The PMCns then reflect the events that occurred after PMC1 became negative and before PMCn becomes negative. This use requires the following MMCR0 bit settings. <ul style="list-style-type: none"> – TRIGGER = 1 – PMC1CE = 0 – PMCnCE = 1 – TBEE = 0 – FCECE = 1 – PMXE = 1 (if a performance monitor interrupt is desired) Resume counting in the PMCns when PMC1 becomes negative, and cause a performance monitor interrupt without freezing any PMCs. The PMCns then reflect the events that occurred between the time PMC1 became negative and the time the interrupt handler reads them. This use requires the following MMCR0 bit settings. <ul style="list-style-type: none"> – TRIGGER = 1 – PMC1CE = 1 – TBEE = 0 – FCECE = 0 – PMXE = 1 The use of the trigger and freeze counter conditions depends on the enabled conditions and events described in Section 10.2, “Performance Monitor Interrupt.”
19–25	PMC1SEL	PMC1 selector. Contains a code (one of at most 128 values) that identifies the event to be counted in PMC1. See Table 10-9 .
26–31	PMC2SEL	PMC2 selector. Contains a code (one of at most 64 values) that identifies the event to be counted in PMC2. See Table 10-10 .

MMCR0 can be accessed with **mtspr** and **mfspir** using SPR 952.

2.1.6.9.2 User Monitor Mode Control Register 0 (UMMCR0)

The contents of MMCR0 are reflected to UMMCR0, which can be read by user-level software. MMCR0 can be accessed with **mfspir** using SPR 936.

2.1.6.9.3 Monitor Mode Control Register 1 (MMCR1)

The monitor mode control register 1 (MMCR1) functions as an event selector for performance monitor counter registers 3, 4, 5, and 6 (PMC3, PMC4, PMC5, PMC6). The MMCR1 register is shown in [Figure 2-31](#).

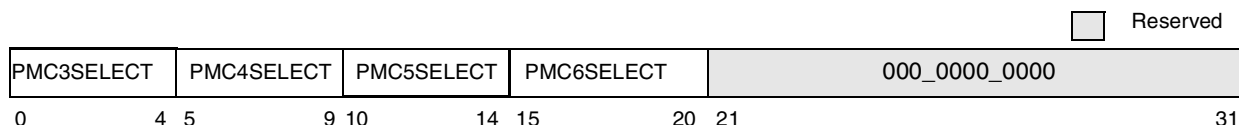


Figure 2-31. Monitor Mode Control Register 1 (MMCR1)

Bit settings for MMCR1 are shown in [Table 2-32](#). The corresponding events are described in [Section 2.1.6.9.8, “Performance Monitor Counter Registers \(PMC1–PMC6\).”](#)

Table 2-32. MMCR1 Field Descriptions

Bits	Name	Description
0–4	PMC3SELECT	PMC3 selector. Contains a code (1 of at most 32 values) that identifies the event to be counted in PMC3. See Table 10-11 .
5–9	PMC4SELECT	PMC4 selector. Contains a code (1 of at most 32 values) that identifies the event to be counted in PMC4. See Table 10-12 .
10–14	PMC5SELECT	PMC5 selector. Contains a code (1 of at most 32 values) that identifies the event to be counted in PMC5. See Table 10-13 .
15–20	PMC6SELECT	PMC6 selector. Contains a code (1 of at most 64 values) that identifies the event to be counted in PMC6. See Table 10-14 .
21–31	—	Reserved

MMCR1 can be accessed with **mtspr** and **mf spr** using SPR 956. User-level software can read the contents of MMCR1 by issuing an **mf spr** instruction to UMMCR1, described in [Section 2.1.6.9.4, “User Monitor Mode Control Register 1 \(UMMCR1\).”](#)

2.1.6.9.4 User Monitor Mode Control Register 1 (UMMCR1)

The contents of MMCR1 are reflected to UMMCR1, which can be read by user-level software. MMCR1 can be accessed with **mf spr** using SPR 940.

2.1.6.9.5 Monitor Mode Control Register 2 (MMCR2)

The monitor mode control register 2 (MMCR2) functions as an event selector for performance monitor counter registers 3 and 4 (PMC3 and PMC4). The MMCR2 register is shown in [Figure 2-32](#).

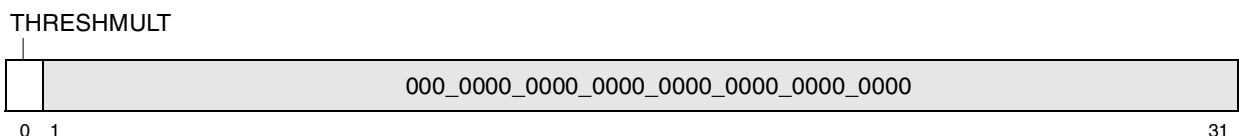


Figure 2-32. Monitor Mode Control Register 2 (MMCR2)

Table 2-33 describes MMCR2 fields.

Table 2-33. MMCR2 Field Descriptions

Bits	Name	Description
0	THRESHMULT	Threshold multiplier. Used to extend the range of the THRESHOLD field, MMCR0[10–15]. 0 Threshold field is multiplied by 2 1 Threshold field is multiplied by 32
1–31	—	Reserved

MMCR2 can be accessed with **mtspr** and **mfspr** using SPR 944. User-level software can read the contents of MMCR2 by issuing an **mfspr** instruction to UMMCR2, described in Section 2.1.6.9.6, “User Monitor Mode Control Register 2 (UMMCR2).”

2.1.6.9.6 User Monitor Mode Control Register 2 (UMMCR2)

The contents of MMCR2 are reflected to UMMCR2, which can be read by user-level software. UMMCR2 can be accessed with the **mfspr** instruction using SPR 928.

2.1.6.9.7 Breakpoint Address Mask Register (BAMR)

The breakpoint address mask register (BAMR), shown in Figure 2-33, is used in conjunction with the events that monitor IABR hits.

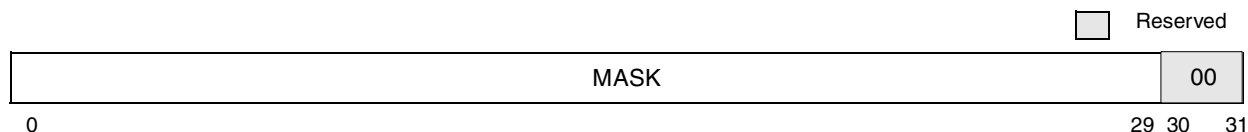


Figure 2-33. Breakpoint Address Mask Register (BAMR)

Table 2-34 describes BAMR fields.

Table 2-34. BAMR Field Descriptions

Bit	Name	Description
0–29	MASK ¹	Used with PMC1 event (PMC1 event 42) that monitor IABR hits. The addresses to be compared for an IABR match are affected by the value in BAMR: <ul style="list-style-type: none"> IABR hit (PMC1, event 42) occurs if IABR_CMP (that is, IABR AND BAMR) = instruction_address_compare (that is, EA AND BAMR) $IABR_CMP[0-29] = IABR[0-29] \text{ AND } BAMR[0-29]$ $instruction_addr_cmp[0-29] = instruction_addr[0-29] \text{ AND } BAMR[0-29]$ Be aware that breakpoint event 42 of PMC1 can be used to trigger performance monitor interrupts when the performance monitor detects an enabled overflow. This feature supports debug purposes and occurs only when IABR[30] is set. To avoid taking one of the above interrupts, make sure that IABR[30] is cleared.
30–31	—	Reserved

¹ A context synchronizing instruction must follow the **mtspr**.

BAMR can be accessed with **mtspr** and **mfspir** using SPR 951. For synchronization requirements on the register see [Section 2.3.2.4, “Synchronization.”](#)

2.1.6.9.8 Performance Monitor Counter Registers (PMC1–PMC6)

PMC1–PMC6, shown in [Figure 2-34](#), are 32-bit counters that can be programmed to generate a performance monitor interrupt when they overflow.

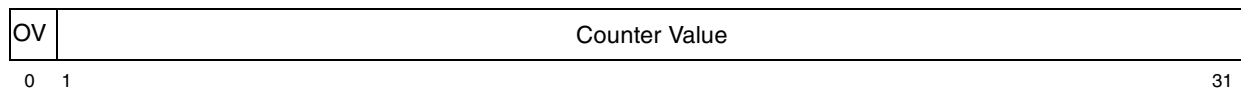


Figure 2-34. Performance Monitor Counter Registers (PMC1–PMC6)

The bits contained in the PMC registers are described in [Table 2-35](#).

Table 2-35. PMC n Field Descriptions

Bits	Name	Description
0	OV	Overflow When this bit is set, it indicates that this counter has overflowed and reached its maximum value so that PMC n [OV] = 1.
1–31	Counter value	Counter value Indicates the number of occurrences of the specified event.

Counters overflow when the high-order (sign) bit becomes set; that is, they reach the value 2,147,483,648 (0x8000_0000). However, an interrupt is not generated unless both MMCR0[PMXE] and either MMCR0[PMC1CE] or MMCR0[PMCCCE] are also set as appropriate.

Note that the interrupt can be masked by clearing MSR[EE]; the performance monitor condition may occur with MSR[EE] cleared, but the interrupt is not taken until MSR[EE] is set. Setting MMCR0[FCECE] forces counters to stop counting when a counter interrupt or any enabled condition or event occurs. Setting MMCR0[TRIGGER] forces counters PMC n ($n > 1$), to begin counting when PMC1 goes negative or an enabled condition or event occurs.

Software is expected to use the **mtspr** instruction to explicitly set PMC to non-overflowed values. Setting an overflowed value may cause an erroneous interrupt. For example, if both MMCR0[PMXE] and either MMCR0[PMC1CE] or MMCR0[PMCCCE] are set and the **mtspr** instruction loads an overflow value, an interrupt may be taken without an event counting having taken place.

The PMC registers can be accessed with the **mtspr** and **mfspir** instructions using the following SPR numbers:

- PMC1 is SPR 953
- PMC2 is SPR 954
- PMC3 is SPR 957
- PMC4 is SPR 958
- PMC5 is SPR 945
- PMC6 is SPR 946

2.1.6.9.9 User Performance Monitor Counter Registers (UPMC1–UPMC6)

The contents of the PMC1–PMC6 are reflected to UPMC1–UPMC6, which can be read by user-level software. The UPMC registers can be read with **mfspir** using the following SPR numbers:

- UPMC1 is SPR 937
- UPMC2 is SPR 938
- UPMC3 is SPR 941
- UPMC4 is SPR 942
- UPMC5 is SPR 929
- UPMC6 is SPR 930

2.1.6.9.10 Sampled Instruction Address Register (SIAR)

The sampled instruction address register (SIAR) is a supervisor-level register that contains the effective address of the last instruction to complete before the performance monitor interrupt is signaled. The SIAR is shown in [Figure 2-35](#).

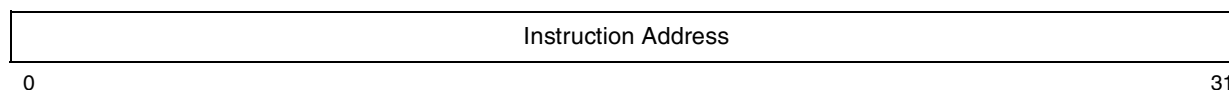


Figure 2-35. Sampled Instruction Address Registers (SIAR)

Note that SIAR is not updated in any of the following conditions:

- Performance monitor counting has been disabled by setting MMCR0[FC].
- Performance monitor interrupt has been disabled by clearing MMCR0[PMXE]

SIAR can be accessed with the **mtspir** and **mfspir** instructions using SPR 955.

2.1.6.9.11 User-Sampled Instruction Address Register (USIAR)

The contents of SIAR are reflected to USIAR, which can be read by user-level software. USIAR can be accessed with the **mfspir** instructions using SPR 939.

2.1.6.9.12 Sampled Data Address Register (SDAR) and User-Sampled Data Address Register (USDAR)

The e600 core does not implement the sampled data address register (SDAR) or the user-level, read-only USDA registers. Note that in previous processors the SDAR and USDAR registers could be written to by boot code without causing an interrupt, this is not the case in the e600 core. A **mtspir** or **mfspir** SDAR or USDAR instruction causes a program interrupt.

2.1.7 Reset Settings

[Table 2-36](#) shows the state of the registers and other resources after a hard reset and before the first instruction is fetched from address 0xFFFF0_0100 (the system reset interrupt vector). When a register is not initialized at hard reset, the setting is undefined.

Table 2-36. Settings Caused by Hard Reset (Used at Power-On)

Resource	Setting
BAMR	0x0000_0000
BATs	Undefined
Caches (L1/L2)	Disabled. The caches are not invalidated and must be invalidated in software before they are enabled.
CR	0x0000_0000
CTR	0x0000_0000
DABR	Breakpoint is disabled. Address is undefined.
DAR	0x0000_0000
DEC	0xFFFF_FFFF
DSISR	0x0000_0000
EAR	0x0000_0000
FPRs	Undefined
FPSCR	0x0000_0000
GPRs	Undefined
HID0	0x8000_0000
HID1	0x000n_n080 (note that bits 14–19 are set to match the settings of <i>pll_cfg</i> [0:5] at reset)
IABR	0x0000_0000 (breakpoint is disabled)
ICTC	0x0000_0000
ICTRL	0x0000_0000
L2CAPTDATAHI	0x0000_0000
L2CAPTDATALO	0x0000_0000
L2CAPTECC	0x0000_0000
L2CR	0x3000_0000
L2ERRADDR	0x0000_0000
L2ERRATTR	0x0000_0000
L2ERRCTL	0x0000_0000
L2ERRDET	0x0000_0000
L2ERRDIS	0x0000_0000
L2ERREADDR	0x0000_0000
L2ERRINJCTL	0x0000_0000
L2ERRINJHI	0x0000_0000
L2ERRINJLO	0x0000_0000
L2ERRINTEN	0x0000_0000
LDSTCR	0x0000_0000

Table 2-36. Settings Caused by Hard Reset (Used at Power-On) (continued)

Resource	Setting
LR	0x0000_0000
MMCR n	0x0000_0000
MSSCR0	0x0000_8000
MSSSR0	0x0000_0000
MSR	0x0000_0040 (only IP set)
S	0x0000_0000
PMC n	Undefined
PTEHI	0x0000_0000
PTELO	0x0000_0000
PVR	
Reservation address	Undefined
Reservation flag	Cleared
SDR1	0x0000_0000
SIAR	0x0000_0000
SPRG0–SPGR7	0x0000_0000
SRs	Undefined
SRR0	0x0000_0000
SRR1	0x0000_0000
TBU and TBL	0x0000_0000
TLBs	Undefined
TLBMISS	0x0000_0000
UMMCR n	0x0000_0000
UPMC n	0x0000_0000
USIAR	0x0000_0000
VRs	Undefined
VRSAVE	0x0000_0000
VSCR	0x0001_0000
XER	0x0000_0000

2.2 Operand Conventions

This section describes the operand conventions as they are represented in two levels of the PowerPC architecture—UISA and VEA. Detailed descriptions are provided of conventions used for storing values in registers and memory, accessing PowerPC registers, and representation of data in these registers.

2.2.1 Floating-Point Execution Models—UISA

The IEEE Std. 754 standard defines conventions for 64- and 32-bit arithmetic. The standard requires that single-precision arithmetic be provided for single-precision operands. The standard permits double-precision arithmetic instructions to have either (or both) single-precision or double-precision operands but states that single-precision arithmetic instructions should not accept double-precision operands.

The PowerPC UISA follows these guidelines:

- Double-precision arithmetic instructions can have single-precision operands but always produce double-precision results.
- Single-precision arithmetic instructions require all operands to be single-precision and always produce single-precision results.

For arithmetic instructions, conversion from double- to single-precision must be done explicitly by software, while conversion from single- to double-precision is done implicitly by the processor.

All implementations of the PowerPC architecture provide the equivalent of the following execution models to ensure that identical results are obtained. The definition of the arithmetic instructions for infinities, denormalized numbers, and NaNs follow conventions described in the following sections.

Although the double-precision format specifies an 11-bit exponent, exponent arithmetic uses two additional bit positions to avoid potential transient overflow conditions. An extra bit is required when denormalized double-precision numbers are prenormalized. A second bit is required to permit computation of the adjusted exponent value in the following examples when the corresponding interrupt enable bit has a value of 1:

- Underflow during multiplication using a denormalized operand
- Overflow during division using a denormalized divisor

2.2.2 Data Organization in Memory and Data Transfers

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

Memory operands can be bytes, half words, words, double words, quad words, or, for the load/store multiple and load/store string instructions, a sequence of bytes or words. The address of a memory operand is the address of its first byte (that is, of its lowest-numbered byte). Operand length is implicit for each instruction.

2.2.3 Alignment and Misaligned Accesses

The operand of a single-register memory access instruction has an alignment boundary equal to its length. An operand's address is misaligned if it is not a multiple of its width.

The concept of alignment is also applied more generally to data in memory. For example, a 12-byte data item is said to be word-aligned if its address is a multiple of four.

Some instructions require their memory operands to have certain alignment. In addition, alignment can affect performance. For single-register memory access instructions, the best performance is obtained when memory operands are aligned.

Instructions are 32 bits (one word) long and must be word-aligned.

The e600 core does not provide hardware support for floating-point memory that is not word-aligned. If a floating-point operand is not word-aligned, the e600 core invokes an alignment interrupt, and it is left up to software to break up the offending memory access operation appropriately. In addition, some non-double-word-aligned memory accesses suffer performance degradation as compared to an aligned access of the same type.

In general, floating-point word accesses should always be word-aligned and floating-point double-word accesses should always be double-word-aligned. Frequent use of misaligned accesses is discouraged because they can degrade overall performance.

2.2.4 Floating-Point Operands

The e600 core provides hardware support for all single- and double-precision floating-point operations for most value representations and all rounding modes. This architecture provides for hardware to implement a floating-point system as defined in ANSI/IEEE Standard 754-1985, *IEEE Standard for Binary Floating Point Arithmetic*. Detailed information about the floating-point execution model can be found in Chapter 3, "Operand Conventions," in the *Programming Environments Manual*.

The e600 core supports non-IEEE Std. 754 mode when FPSCR[29] is set. In this mode, denormalized numbers are treated in a non-IEEE Std. 754 conforming manner. This is accomplished by delivering results that are forced to the value zero.

2.3 Instruction Set Summary

This chapter describes instructions and addressing modes defined for the e600 core. These instructions are divided into the following functional categories:

- Integer instructions—These include arithmetic and logical instructions. For more information, see [Section 2.3.4.1, “Integer Instructions.”](#)
- Floating-point instructions—These include floating-point arithmetic instructions, as well as instructions that affect the floating-point status and control register (FPSCR). For more information, see [Section 2.3.4.2, “Floating-Point Instructions.”](#)

Load and store instructions—These include integer and floating-point load and store instructions. For more information, see [Section 2.3.4.3, “Load and Store Instructions.”](#)

- Flow control instructions—These include branching instructions, condition register logical instructions, trap instructions, and other instructions that affect the instruction flow. For more information, see [Section 2.3.4.4, “Branch and Flow Control Instructions.”](#)
- Processor control instructions—These instructions are used for synchronizing memory accesses and managing segment registers. For more information, see [Section 2.3.4.6, “Processor Control Instructions—UISA,”](#) [Section 2.3.5.1, “Processor Control Instructions—VEA,”](#) and [Section 2.3.6.2, “Processor Control Instructions—OEA.”](#)
- Memory synchronization instructions—These instructions are used for memory synchronizing. See [Section 2.3.4.7, “Memory Synchronization Instructions—UISA,”](#) and [Section 2.3.5.2, “Memory Synchronization Instructions—VEA,”](#) for more information.
- Memory control instructions—These instructions provide control of caches and TLBs. For more information, see [Section 2.3.5.3, “Memory Control Instructions—VEA,”](#) and [Section 2.3.6.3, “Memory Control Instructions—OEA.”](#)
- External control instructions—These include instructions for use with special input/output devices. For more information, see [Section 2.3.5.4, “Optional External Control Instructions.”](#)
- AltiVec instructions—AltiVec technology does not have optional instructions defined, so all instructions listed in the *AltiVec Technology Programming Environments Manual* are implemented for the e600 core. Instructions that are implementation specific are described in [Section 2.6.2, “AltiVec Instructions with Specific Implementations for the e600 Core.”](#)

Note that this grouping of instructions does not necessarily indicate the execution unit that processes a particular instruction or group of instructions. This information, which is useful for scheduling instructions most effectively, is provided in [Chapter 6, “Instruction Timing.”](#)

Integer instructions operate on word operands. Floating-point instructions operate on single-precision and double-precision floating-point operands. AltiVec instructions operate on byte, half-word, word, and quad-word operands. The PowerPC architecture uses instructions that are 4 bytes long and word-aligned. It provides for byte, half-word, and word operand loads and stores between memory and a set of 32 general-purpose registers (GPRs). It provides for word and double-word operand loads and stores between memory and a set of 32 floating-point registers (FPRs). It also provides for byte, half-word, word, and quad-word operand loads and stores between memory and a set of 32 vector registers (VRs).

Arithmetic and logical instructions do not read or modify memory. To use the contents of a memory location in a computation and then modify the same or another memory location, the memory contents must be loaded into a register, modified, and then written to the target location using load and store instructions.

The description of each instruction includes the mnemonic and a formatted list of operands. To simplify assembly language programming, a set of simplified mnemonics and symbols is provided for some of the frequently-used instructions; see Appendix F, “Simplified Mnemonics,” in the *Programming Environments Manual* for a complete list of simplified mnemonics. Programs written to be portable across the various assemblers for the PowerPC architecture should not assume the existence of mnemonics not described in that document.

2.3.1 Classes of Instructions

The e600 core instructions belong to one of the following three classes:

- Defined
- Illegal
- Reserved

Note that while the definitions of these terms are consistent among the processors that implement the PowerPC architecture, the assignment of these classifications is not. For example, PowerPC instructions defined for 64-bit implementations are treated as illegal by 32-bit implementations such as the e600 core.

The class is determined by examining the primary opcode and the extended opcode, if any. If the opcode, or combination of opcode and extended opcode, is not that of a defined instruction or of a reserved instruction, the instruction is illegal.

Instruction encodings that are now illegal can become assigned to instructions in the architecture or can be reserved by being assigned to processor-specific instructions.

2.3.1.1 Definition of Boundedly Undefined

If instructions are encoded with incorrectly set bits in reserved fields, the results on execution can be said to be boundedly undefined. If a user-level program executes the incorrectly coded instruction, the resulting undefined results are bounded in that a spurious change from user to supervisor state is not allowed, and the level of privilege exercised by the program in relation to memory access and other system resources cannot be exceeded. Boundedly undefined results for a given instruction can vary between implementations and between execution attempts in the same implementation.

2.3.1.2 Defined Instruction Class

Defined instructions are guaranteed to be supported in all implementations of the PowerPC architecture, except as stated in the instruction descriptions in Chapter 8, “Instruction Set,” of the *Programming Environments Manual*. The e600 core provides hardware support for all instructions defined for 32-bit implementations. It does not support the optional **fsqrt**, **fsqrts**, and **tlbia** instructions.

A processor invokes the illegal instruction error handler (part of the program interrupt) when it encounters a PowerPC instruction that has not been implemented. The instruction can be emulated in software, as required.

A defined instruction can have invalid forms. The e600 core provides limited support for instructions represented in an invalid form.

2.3.1.3 Illegal Instruction Class

Illegal instructions can be grouped into the following categories:

- Instructions not defined in the PowerPC architecture. The following primary opcodes are defined as illegal, but can be used in future extensions to the architecture:

1, 5, 6, 9, 22, 56, 57, 60, 61

Future versions of the PowerPC architecture can define any of these instructions to perform new functions.

- Instructions defined in the PowerPC architecture but not implemented in a specific implementation. For example, instructions that can be executed on 64-bit processors that implement the PowerPC architecture are considered illegal by 32-bit processors such as the e600 core.

The following primary opcodes are defined for 64-bit implementations only and are illegal on the e600 core:

2, 30, 58, 62

- All unused extended opcodes are illegal. The unused extended opcodes can be determined from information in [Section A.3, “Instructions Sorted by Mnemonic \(Binary\),”](#) and [Section 2.3.1.4, “Reserved Instruction Class.”](#) Notice that extended opcodes for instructions defined only for 64-bit implementations are illegal in 32-bit implementations, and vice versa. The following primary opcodes have unused extended opcodes:

17, 19, 31, 59, 63 (Primary opcodes 30 and 62 are illegal for all 32-bit implementations, but as 64-bit opcodes, they have some unused extended opcodes.)

- An instruction consisting of only zeros is guaranteed to be an illegal instruction. This increases the probability that an attempt to execute data or memory that was not initialized invokes the system illegal instruction error handler (a program interrupt). Note that if only the primary opcode consists of all zeros, the instruction is considered a reserved instruction, as described in [Section 2.3.1.4, “Reserved Instruction Class.”](#)

The e600 core invokes the system illegal instruction error handler (a program interrupt) when it detects any instruction from this class or any instructions defined only for 64-bit implementations.

See [Section 4.6.7, “Program Interrupt \(0x00700\),”](#) for additional information about illegal and invalid instruction interrupts. Except for an instruction consisting of binary zeros, illegal instructions are available for additions to the PowerPC architecture.

2.3.1.4 Reserved Instruction Class

Reserved instructions are allocated to specific implementation-dependent purposes not defined by the PowerPC architecture. Attempting to execute a reserved instruction that has not been implemented invokes the illegal instruction error handler (a program interrupt). See “Program Interrupt (0x0_0700),” in Chapter 6, “Interrupts,” in the *Programming Environments Manual* for information about illegal and invalid instruction interrupts.

The PowerPC architecture defines four types of reserved instructions:

- Instructions in the POWER architecture not part of the PowerPC UISA. For details on POWER architecture incompatibilities and how they are handled by processors that implement the PowerPC architecture, see Appendix B, “POWER Architecture Cross Reference,” in the *Programming Environments Manual*.
- Implementation-specific instructions required for the processor to conform to the PowerPC architecture (none of these are implemented in the e600 core)
- All other implementation-specific instructions
- Architecturally allowed extended opcodes

2.3.2 Addressing Modes

This section provides an overview of conventions for addressing memory and for calculating effective addresses as defined by the PowerPC architecture for 32-bit implementations. For more detailed information, see “Conventions,” in Chapter 4, “Addressing Modes and Instruction Set Summary,” of the *Programming Environments Manual*.

2.3.2.1 Memory Addressing

A program references memory using the effective (logical) address computed by the processor when it executes a memory access or branch instruction or when it fetches the next sequential instruction.

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

2.3.2.2 Memory Operands

Memory operands can be bytes, half words, words, double words, quad words or, for the load/store multiple and load/store string instructions, a sequence of bytes or words. The address of a memory operand is the address of its first byte (that is, of its lowest-numbered byte). Operand length is implicit for each instruction. The PowerPC architecture supports both big-endian and little-endian byte ordering. The default byte and bit ordering is big endian. See “Byte Ordering,” in Chapter 3, “Operand Conventions,” of the *Programming Environments Manual* for more information about big- and little-endian byte ordering.

The operand of a single-register memory access instruction has a natural alignment boundary equal to the operand length; that is, the natural address of an operand is an integral multiple of its length. A memory operand is said to be aligned if it is aligned at its natural boundary; otherwise it is misaligned. For a detailed discussion about memory operands, see Chapter 3, “Operand Conventions,” of the *Programming Environments Manual*.

2.3.2.3 Effective Address Calculation

An effective address is the 32-bit sum computed by the processor when executing a memory access or branch instruction or when fetching the next sequential instruction. For a memory access instruction, if the sum of the effective address and the operand length exceeds the maximum effective address, the memory operand is considered to wrap around from the maximum effective address through effective address 0, as described in the following paragraphs.

Effective address computations for both data and instruction accesses use 32-bit unsigned binary arithmetic. A carry from bit 0 is ignored.

Load and store operations have the following modes of effective address generation:

- $EA = (rA|0) + \text{offset}$ (including offset = 0) (register indirect with immediate index)
- $EA = (rA|0) + rB$ (register indirect with index)

Refer to [Section 2.3.4.3.2, “Integer Load and Store Address Generation,”](#) for a detailed description of effective address generation for load and store operations.

Branch instructions have three categories of effective address generation:

- Immediate
- Link register indirect
- Count register indirect

2.3.2.4 Synchronization

The synchronization described in this section refers to the state of the processor that is performing the synchronization.

2.3.2.4.1 Context Synchronization

The System Call (**sc**) and Return from Interrupt (**rfi**) instructions perform context synchronization by allowing previously issued instructions to complete before performing a change in context. Execution of one of these instructions ensures the following:

- No higher priority interrupt exists (**sc**).
- All previous instructions have completed to a point where they can no longer cause an interrupt. If a prior memory access instruction causes direct-store error interrupts, the results are guaranteed to be determined before this instruction is executed.
- Previous instructions complete execution in the context (privilege, protection, and address translation) under which they were issued.
- The instructions following the **sc** or **rfi** instruction execute in the context established by these instructions.

Modifying certain registers requires software synchronization to follow certain register dependencies. [Table 2-37](#) defines specific synchronization procedures that are required when using various SPRs and specific bits within SPRs. Context synchronizing instructions that can be used are: **isync**, **sc**, **rfi**, and any interrupt other than system reset and machine check. If multiple bits are being modified that have different synchronization requirements, the most restrictive requirements can be used. However, a **mtspr**

instruction to modify either HID0[ICE] or HID0[ICFI] should not also modify other HID0 bits that requires synchronization.

Table 2-37. Control Registers Synchronization Requirements

Register	Bits	Synchronization Requirements
BAMR	Any	A context synchronizing instruction must follow the mtspr .
DABR	Any	A dssall and sync must precede the mtspr and then a sync and a context synchronizing instruction must follow. Note that if a user is not using the AltiVec data streaming instructions, then a dssall is not necessary prior to accessing the register.
DBATs	Any	A dssall and sync must precede the mtspr and then a sync and a context synchronizing instruction must follow. Note that if a user is not using the AltiVec data streaming instructions, then a dssall is not necessary prior to accessing the register.
EAR	Any	A dssall and sync must precede the mtspr and then a sync and a context synchronizing instruction must follow. Note that if a user is not using the AltiVec data streaming instructions, then a dssall is not necessary prior to accessing register.

Table 2-37. Control Registers Synchronization Requirements (continued)

Register	Bits	Synchronization Requirements
HID0	BHTCLR	A context synchronizing instruction must precede an mtspr and a branch instruction should follow. The branch instruction may be either conditional or unconditional. It ensures that all subsequent branch instructions see the newly initialized BHT values. For correct results, the BHT should be disabled (HID0[BHT] = 0) before setting BHTCLR.
	BHT	A context synchronizing instruction must follow the mtspr .
	BTIC	
	DPM	
	FOLD	
	LRSTK	
	NAP	
	NHR	
	SLEEP	
	SPD	
	TBEN	
	DCE	A dssall and sync must precede an mtspr and then a sync and context synchronizing instruction must follow. Note that if a user is not using the AltiVec data streaming instructions, then a dssall is not necessary prior to accessing the HID0[DCE] or HID0[DCFI] bit.
	DCFI	
	DLOCK	
	NOPDST	
	STEN	
	ICE	A context synchronizing instruction must immediately follow an mtspr . An mtspr instruction for HID0 should not modify either of these bits at the same time it modifies another bit that requires additional synchronization.
	ICFI	
	ILOCK	A context synchronizing instruction must precede and follow an mtspr .
	NOPTI	An mtspr must follow a sync and a context synchronizing instruction.
SGE		
XAEN	A dssall and sync must precede an mtspr and then a sync and a context-synchronizing instruction must follow. Alteration of HID0[XAEN] must be done with caches and translation disabled. The caches and TLBs must be flushed before they are re-enabled after the XAEN bit is altered. Note that if a user is not using the AltiVec data streaming instructions, then a dssall is not necessary prior to accessing the HID0[XAEN] bit.	
HID1	Any	A sync and context synchronizing instruction must follow an mtspr .
IABR	Any	A context synchronizing instruction must follow an mtspr .
IBATs	Any	A context synchronizing instruction must follow an mtspr .

Table 2-37. Control Registers Synchronization Requirements (continued)

Register	Bits	Synchronization Requirements
ICTRL	EDCE	A dssall and sync must precede an mtspr and then a sync and context synchronizing instruction must follow. Note that if a user is not using the AltiVec data streaming instructions, then a dssall is not necessary prior to accessing the ICTRL[EDCE] bit.
	ICWL	A context synchronizing instruction must precede and follow an mtspr .
	EICE	
L2ERRDIS	Any	A sync must precede an mtspr and then a sync and isync must follow.
LDSTCR	Any	A dssall and sync must precede an mtspr and then a sync and context synchronizing instruction must follow. Note that if a user is not using the AltiVec data streaming instructions, then a dssall is not necessary prior to accessing the register.
MSR	BE	A context synchronizing instruction must follow an mtmsr instruction.
	VEC	
	FE0	
	FE1	
	FP	
	SE	
	IR	A context synchronizing instruction must follow an mtmsr . When changing the MSR[IR] bit the context synchronizing instruction must reside at both the untranslated and the translated address following the mtmsr .
	DR	A dssall and sync must precede an mtmsr and then a sync and context synchronizing instruction must follow. Note that if a user is not using the AltiVec data streaming instructions, then a dssall is not necessary prior to accessing the MSR[DR] or MSR[PR] bit.
	PR	
LE	A dssall and sync must precede an rfi to guarantee a solid context boundary. Note that if a user is not using AltiVec data streaming instructions, a dssall is not necessary before accessing MSR[LE].	
	POW	A dssall and sync must precede an mtmsr instruction and then a context synchronizing instruction must follow.
MSSCR0	Any	A dssall and sync must precede an mtspr instruction and then a sync and context synchronizing instruction must follow. Note that if a user is not using the AltiVec data streaming instructions, then a dssall is not necessary prior to accessing the register.
SDR1	Any	A dssall and sync must precede an mtspr and then a sync and context synchronizing instruction must follow. Note that if a user is not using the AltiVec data streaming instructions, then a dssall is not necessary prior to accessing the register.
SR0–SR15	Any	A dssall and sync must precede an mtsr or mtsrin instruction and then a sync and context synchronizing instruction must follow. Note that if a user is not using the AltiVec data streaming instructions, then a dssall is not necessary prior to accessing the register.
Other registers or bits	—	No special synchronization requirements.

2.3.2.4.2 Execution Synchronization

An instruction is execution synchronizing if all previously initiated instructions appear to have completed before the instruction is initiated or, in the case of **sync** and **isync**, before the instruction completes. For example, the Move to Machine State Register (**mtmsr**) instruction is execution synchronizing. It ensures that all preceding instructions have completed execution and cannot cause an interrupt before the instruction executes, but does not ensure subsequent instructions execute in the newly established environment. For example, if the **mtmsr** sets the MSR[PR] bit, unless an **isync** immediately follows the **mtmsr** instruction, a privileged instruction could be executed or privileged access could be performed without causing an interrupt even though MSR[PR] indicates user mode.

2.3.2.4.3 Instruction-Related Interrupts

There are two kinds of interrupts in the e600 core—those caused directly by the execution of an instruction and those caused by an asynchronous event. Either can cause components of the system software to be invoked.

Interrupts can be caused directly by the execution of an instruction as follows:

- An attempt to execute an illegal instruction causes the illegal instruction (program interrupt) handler to be invoked. An attempt by a user-level program to execute the supervisor-level instructions listed below causes the privileged instruction (program interrupt) handler to be invoked. The e600 core provides the following supervisor-level instructions—**dcbi**, **mfmsr**, **mf spr**, **mfsr**, **mfsrin**, **mtmsr**, **mtspr**, **mts**, **mts rin**, **r fi**, **tlbie**, and **tlbsync**. Note that the privilege level of the **mf spr** and **mtspr** instructions depends on the SPR encoding.
- Any **mtspr**, **mf spr**, or **mftb** instruction with an invalid SPR (or TBR) field causes an illegal type program interrupt. Likewise, a program interrupt is taken if user-level software tries to access a supervisor-level SPR. An **mtspr** instruction executing in supervisor mode (MSR[PR] = 0) with the SPR field specifying PVR (read-only register) executes as a no-op.
- An attempt to access memory that is not available (page fault) causes the ISI or DSI interrupt handler to be invoked.
- The execution of an **sc** instruction invokes the system call interrupt handler that permits a program to request the system to perform a service.
- The execution of a trap instruction invokes the program interrupt trap handler.
- The execution of an instruction that causes a floating-point exception while exceptions are enabled in the MSR invokes the program interrupt handler.

A detailed description of exception conditions is provided in [Chapter 4, “Interrupts.”](#)

2.3.3 Instruction Set Overview

This section provides a brief overview of the PowerPC instructions implemented in the e600 core and highlights any special information with respect to how the e600 core implements a particular instruction. Note that the categories used in this section correspond to those used in Chapter 4, “Addressing Modes and Instruction Set Summary,” in the *Programming Environments Manual*. These categorizations are somewhat arbitrary, are provided for the convenience of the programmer, and do not necessarily reflect the PowerPC architecture specification.

Note that some instructions have the following optional features:

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

2.3.4 PowerPC UISA Instructions

The PowerPC UISA includes the base user-level instruction set (excluding a few user-level cache control, synchronization, and time base instructions), user-level registers, programming model, data types, and addressing modes. This section discusses the instructions defined in the UISA.

2.3.4.1 Integer Instructions

This section describes the integer instructions. These consist of the following:

- Integer arithmetic instructions
- Integer compare instructions
- Integer logical instructions
- Integer rotate and shift instructions

Integer instructions use the content of the GPRs as source operands and place results into GPRs, the XER register, and condition register (CR) fields.

2.3.4.1.1 Integer Arithmetic Instructions

Table 2-38 lists the integer arithmetic instructions for the processors that implement the PowerPC architecture.

Table 2-38. Integer Arithmetic Instructions

Name	Mnemonic	Syntax
Add Immediate	addi	rD,rA,SIMM
Add Immediate Shifted	addis	rD,rA,SIMM
Add	add (add. addo addo.)	rD,rA,rB
Subtract From	subf (subf. subfo subfo.)	rD,rA,rB
Add Immediate Carrying	addic	rD,rA,SIMM
Add Immediate Carrying and Record	addic.	rD,rA,SIMM
Subtract from Immediate Carrying	subfic	rD,rA,SIMM
Add Carrying	addc (addc. addco addco.)	rD,rA,rB
Subtract from Carrying	subfc (subfc. subfco subfco.)	rD,rA,rB
Add Extended	adde (adde. addeo addeo.)	rD,rA,rB
Subtract from Extended	subfe (subfe. subfeo subfeo.)	rD,rA,rB
Add to Minus One Extended	addme (addme. addmeo addmeo.)	rD,rA
Subtract from Minus One Extended	subfme (subfme. subfmeo subfmeo.)	rD,rA

Table 2-38. Integer Arithmetic Instructions (continued)

Name	Mnemonic	Syntax
Add to Zero Extended	addze (addze. addzeo addzeo.)	rD,rA
Subtract from Zero Extended	subfze (subfze. subfzeo subfzeo.)	rD,rA
Negate	neg (neg. nego nego.)	rD,rA
Multiply Low Immediate	mulli	rD,rA,SIMM
Multiply Low Word	mullw (mullw. mullwo mullwo.)	rD,rA,rB
Multiply High Word	mulhw (mulhw.)	rD,rA,rB
Multiply High Word Unsigned	mulhwu (mulhwu.)	rD,rA,rB
Divide Word	divw (divw. divwo divwo.)	rD,rA,rB
Divide Word Unsigned	divwu (divwu. divwuo divwuo.)	rD,rA,rB

Although there is no Subtract Immediate instruction, its effect can be achieved by using an **addi** instruction with the immediate operand negated. Simplified mnemonics are provided that include this negation. The **subf** instructions subtract the second operand (**rA**) from the third operand (**rB**). Simplified mnemonics are provided in which the third operand is subtracted from the second operand. See Appendix F, “Simplified Mnemonics,” in the *Programming Environments Manual* for examples.

The UISA states that an implementation that executes instructions that set the overflow enable bit (OE) or the carry bit (CA) can either execute these instructions slowly or prevent execution of the subsequent instruction until the operation completes. Chapter 6, “Instruction Timing,” describes how the e600 core handles CR dependencies. The summary overflow bit (SO) and overflow bit (OV) in the XER register are set to reflect an overflow condition of a 32-bit result. This can happen only when OE = 1.

2.3.4.1.2 Integer Compare Instructions

The integer compare instructions algebraically or logically compare the contents of register **rA** with either the zero-extended value of the UIMM operand, the sign-extended value of the SIMM operand, or the contents of **rB**. The comparison is signed for the **cmpi** and **cmp** instructions, and unsigned for the **cmpli** and **cmpl** instructions. Table 2-39 summarizes the integer compare instructions.

Table 2-39. Integer Compare Instructions

Name	Mnemonic	Syntax
Compare Immediate	cmpi	crfD,L,rA,SIMM
Compare	cmp	crfD,L,rA,rB
Compare Logical Immediate	cmpli	crfD,L,rA,UIMM
Compare Logical	cmpl	crfD,L,rA,rB

The **crfD** operand can be omitted if the result of the comparison is to be placed in CR0. Otherwise the target CR field must be specified in **crfD**, using an explicit field number.

For information on simplified mnemonics for the integer compare instructions see Appendix F, “Simplified Mnemonics,” in the *Programming Environments Manual*.

2.3.4.1.3 Integer Logical Instructions

The logical instructions shown in Table 2-40 perform bit-parallel operations on the specified operands. Logical instructions with the CR updating enabled (uses dot suffix) and instructions **andi.** and **andis.** set CR field CR0 to characterize the result of the logical operation. Logical instructions do not affect XER[SO], XER[OV], or XER[CA].

See Appendix F, “Simplified Mnemonics,” in the *Programming Environments Manual* for simplified mnemonic examples for integer logical operations.

Table 2-40. Integer Logical Instructions

Name	Mnemonic	Syntax	Implementation Notes
AND Immediate	andi.	rA,rS,UIMM	—
AND Immediate Shifted	andis.	rA,rS,UIMM	—
OR Immediate	ori	rA,rS,UIMM	The PowerPC architecture defines ori r0,r0,0 as the preferred form for the no-op instruction. The dispatcher discards this instruction and only dispatches it to the completion queue, but not to any execution unit.
OR Immediate Shifted	oris	rA,rS,UIMM	—
XOR Immediate	xori	rA,rS,UIMM	—
XOR Immediate Shifted	xoris	rA,rS,UIMM	—
AND	and (and.)	rA,rS,rB	—
OR	or (or.)	rA,rS,rB	—
XOR	xor (xor.)	rA,rS,rB	—
NAND	nand (nand.)	rA,rS,rB	—
NOR	nor (nor.)	rA,rS,rB	—
Equivalent	eqv (eqv.)	rA,rS,rB	—
AND with Complement	andc (andc.)	rA,rS,rB	—
OR with Complement	orc (orc.)	rA,rS,rB	—
Extend Sign Byte	extsb (extsb.)	rA,rS	—
Extend Sign Half Word	extsh (extsh.)	rA,rS	—
Count Leading Zeros Word	cntlzw (cntlzw.)	rA,rS	—

2.3.4.1.4 Integer Rotate and Shift Instructions

Rotation operations are performed on data from a GPR, and the result, or a portion of the result, is returned to a GPR. See Appendix F, “Simplified Mnemonics,” in the *Programming Environments Manual* for a complete list of simplified mnemonics that allows simpler coding of often-used functions such as clearing

the leftmost or rightmost bits of a register, left justifying or right justifying an arbitrary field, and simple rotates and shifts.

Integer rotate instructions rotate the contents of a register. The result of the rotation is either inserted into the target register under control of a mask (if a mask bit is 1 the associated bit of the rotated data is placed into the target register, and if the mask bit is 0 the associated bit in the target register is unchanged), or ANDed with a mask before being placed into the target register.

The integer rotate instructions are summarized in [Table 2-41](#).

Table 2-41. Integer Rotate Instructions

Name	Mnemonic	Syntax
Rotate Left Word Immediate then AND with Mask	rlwinm (rlwinm.)	rA,rS,SH,MB,ME
Rotate Left Word then AND with Mask	rlwnm (rlwnm.)	rA,rS,rB,MB,ME
Rotate Left Word Immediate then Mask Insert	rlwimi (rlwimi.)	rA,rS,SH,MB,ME

The integer shift instructions perform left and right shifts. Immediate-form logical (unsigned) shift operations are obtained by specifying masks and shift values for certain rotate instructions. Simplified mnemonics (shown in Appendix F, “Simplified Mnemonics,” in the *Programming Environments Manual*) are provided to make coding of such shifts simpler and easier to understand.

Multiple-precision shifts can be programmed as shown in Appendix C, “Multiple-Precision Shifts,” in the *Programming Environments Manual*. The integer shift instructions are summarized in [Table 2-42](#).

Table 2-42. Integer Shift Instructions

Name	Mnemonic	Syntax
Shift Left Word	slw (slw.)	rA,rS,rB
Shift Right Word	srw (srw.)	rA,rS,rB
Shift Right Algebraic Word Immediate	srawi (srawi.)	rA,rS,SH
Shift Right Algebraic Word	sraw (sraw.)	rA,rS,rB

2.3.4.2 Floating-Point Instructions

This section describes the floating-point instructions, which include the following:

- Floating-point arithmetic instructions
- Floating-point multiply-add instructions
- Floating-point rounding and conversion instructions
- Floating-point compare instructions
- Floating-point status and control register instructions
- Floating-point move instructions

See [Section 2.3.4.3, “Load and Store Instructions,”](#) for information about floating-point loads and stores.

The PowerPC architecture supports a floating-point system as defined in IEEE Std. 754, but requires software support to conform with that standard. All floating-point operations conform to IEEE Std.754, except if software sets the non-IEEE mode bit (FPSCR[NI]).

2.3.4.2.1 Floating-Point Arithmetic Instructions

The floating-point arithmetic instructions are summarized in [Table 2-43](#).

Table 2-43. Floating-Point Arithmetic Instructions

Name	Mnemonic	Syntax
Floating Add (Double-Precision)	fadd (fadd.)	frD,frA,frB
Floating Add Single	fadds (fadds.)	frD,frA,frB
Floating Subtract (Double-Precision)	fsub (fsub.)	frD,frA,frB
Floating Subtract Single	fsubs (fsubs.)	frD,frA,frB
Floating Multiply (Double-Precision)	fmul (fmul.)	frD,frA,frC
Floating Multiply Single	fmuls (fmuls.)	frD,frA,frC
Floating Divide (Double-Precision)	fdiv (fdiv.)	frD,frA,frB
Floating Divide Single	fdivs (fdivs.)	frD,frA,frB
Floating Reciprocal Estimate Single ¹	fres (fres.)	frD,frB
Floating Reciprocal Square Root Estimate ¹	frsqrte (frsqrte.)	frD,frB
Floating Select ¹	fsel	frD,frA,frC,frB

¹ These instructions are optional in the PowerPC architecture.

All single-precision arithmetic instructions are performed using a double-precision format. The floating-point architecture is a single-pass implementation for double-precision products. In most cases, a single-precision instruction using only single-precision operands, in double-precision format, has the same latency as its double-precision equivalent.

2.3.4.2.2 Floating-Point Multiply-Add Instructions

These instructions combine multiply and add operations without an intermediate rounding operation. The floating-point multiply-add instructions are summarized in [Table 2-44](#).

Table 2-44. Floating-Point Multiply-Add Instructions

Name	Mnemonic	Syntax
Floating Multiply-Add (Double-Precision)	fmadd (fmadd.)	frD,frA,frC,frB
Floating Multiply-Add Single	fmadds (fmadds.)	frD,frA,frC,frB
Floating Multiply-Subtract (Double-Precision)	fmsub (fmsub.)	frD,frA,frC,frB
Floating Multiply-Subtract Single	fmsubs (fmsubs.)	frD,frA,frC,frB
Floating Negative Multiply-Add (Double-Precision)	fnmadd (fnmadd.)	frD,frA,frC,frB
Floating Negative Multiply-Add Single	fnmadds (fnmadds.)	frD,frA,frC,frB

Table 2-44. Floating-Point Multiply-Add Instructions (continued)

Name	Mnemonic	Syntax
Floating Negative Multiply-Subtract (Double-Precision)	fnmsub (fnmsub.)	frD,frA,frC,frB
Floating Negative Multiply-Subtract Single	fnmsubs (fnmsubs.)	frD,frA,frC,frB

2.3.4.2.3 Floating-Point Rounding and Conversion Instructions

The Floating Round to Single-Precision (**frsp**) instruction is used to truncate a 64-bit double-precision number to a 32-bit single-precision floating-point number. The floating-point convert instructions convert a 64-bit double-precision floating-point number to a 32-bit signed integer number.

Examples of uses of these instructions to perform various conversions can be found in Appendix D, “Floating-Point Models,” in the *Programming Environments Manual*.

Table 2-45. Floating-Point Rounding and Conversion Instructions

Name	Mnemonic	Syntax
Floating Round to Single	frsp (frsp.)	frD,frB
Floating Convert to Integer Word	fctiw (fctiw.)	frD,frB
Floating Convert to Integer Word with Round toward Zero	fctiwz (fctiwz.)	frD,frB

2.3.4.2.4 Floating-Point Compare Instructions

Floating-point compare instructions compare the contents of two floating-point registers. The comparison ignores the sign of zero (that is $+0 = -0$). The floating-point compare instructions are summarized in [Table 2-46](#).

Table 2-46. Floating-Point Compare Instructions

Name	Mnemonic	Syntax
Floating Compare Unordered	fcmpu	crfD,frA,frB
Floating Compare Ordered	fcmpo	crfD,frA,frB

2.3.4.2.5 Floating-Point Status and Control Register Instructions

Every FPSCR instruction appears to synchronize the effects of all floating-point instructions executed by a given processor. Executing an FPSCR instruction ensures that all floating-point instructions previously initiated by the given processor appear to have completed before the FPSCR instruction is initiated and that no subsequent floating-point instructions appear to be initiated by the given processor until the FPSCR instruction has completed. The FPSCR instructions are summarized in [Table 2-47](#).

Table 2-47. Floating-Point Status and Control Register Instructions

Name	Mnemonic	Syntax
Move from FPSCR	mffs (mffs.)	frD
Move to Condition Register from FPSCR	mcrfs	crfD,crfS

Table 2-47. Floating-Point Status and Control Register Instructions (continued)

Name	Mnemonic	Syntax
Move to FPSCR Field Immediate	mtfsfi (<i>mtfsfi.</i>)	crfD , IMM
Move to FPSCR Fields	mtfsf (<i>mtfsf.</i>)	FM, frB
Move to FPSCR Bit 0	mtfsb0 (<i>mtfsb0.</i>)	crbD
Move to FPSCR Bit 1	mtfsb1 (<i>mtfsb1.</i>)	crbD

Implementation Note—The PowerPC architecture states that in some implementations, the Move to FPSCR Fields (**mtfsf**) instruction can perform more slowly when only some of the fields are updated as opposed to all of the fields. In the e600 core, there is no degradation of performance.

2.3.4.2.6 Floating-Point Move Instructions

Floating-point move instructions copy data from one FPR to another. The floating-point move instructions do not modify the FPSCR. The CR update option in these instructions controls the placing of result status into CR1. [Table 2-48](#) summarizes the floating-point move instructions.

Table 2-48. Floating-Point Move Instructions

Name	Mnemonic	Syntax
Floating Move Register	fmr (<i>fmr.</i>)	frD , frB
Floating Negate	fneg (<i>fneg.</i>)	frD , frB
Floating Absolute Value	fabs (<i>fabs.</i>)	frD , frB
Floating Negative Absolute Value	fnabs (<i>fnabs.</i>)	frD , frB

2.3.4.3 Load and Store Instructions

Load and store instructions are issued and translated in program order; however, the accesses can occur out of order. Synchronizing instructions are provided to enforce strict ordering. This section describes the load and store instructions, which consist of the following:

- Integer load instructions
- Integer store instructions
- Integer load and store with byte-reverse instructions
- Integer load and store multiple instructions
- Floating-point load instructions
- Floating-point store instructions
- Memory synchronization instructions

Implementation Note—The following describes how the e600 core handles misalignment:

The e600 core provides hardware support for misaligned memory accesses. It performs those accesses within a single cycle if the operand lies within a double-word boundary. Misaligned memory accesses that cross a double-word boundary degrade performance.

Although many misaligned memory accesses are supported in hardware, the frequent use of them is discouraged because they can compromise the overall performance of the processor. Only one outstanding misalignment at a time is supported which means it is non-pipelined.

Accesses that cross a translation boundary can be restarted. That is, a misaligned access that crosses a page boundary is completely restarted if the second portion of the access causes a page fault. This can cause the first access to be repeated.

On some processors, such as the MPC603e, a TLB reload operation causes an instruction restart. On the e600 core, TLB reloads are performed transparently (if hardware table search operations are enabled—HID0[STEN] = 0) and only a page fault causes a restart. If software table searching is enabled (HID0[STEN] = 1) on the e600 core, a TLB miss causes an instruction restart (as it causes a TLB miss interrupt)

2.3.4.3.1 Self-Modifying Code

When a processor modifies a memory location that can be contained in the instruction cache, software must ensure that memory updates are visible to the instruction fetching mechanism. This can be achieved by executing the following instruction sequence (using either **dcbst** or **dcbf**):

```

dcbst (or dcbf) | update memory
sync           | wait for update
icbi          | remove (invalidate) copy in instruction cache
sync          | ensure that ICBI invalidate at the icache has completed
isync         | remove copy in own instruction buffer
    
```

These operations are required because the data cache is a write-back cache. Because instruction fetching bypasses the data cache, changes to items in the data cache can not be reflected in memory until the fetch operations complete. The **sync** after the **icbi** is required to ensure that the **icbi** invalidation has completed in the instruction cache.

Special care must be taken to avoid coherency paradoxes in systems that implement unified secondary caches (like the e600 core), and designers should carefully follow the guidelines for maintaining cache coherency that are provided in the VEA, and discussed in Chapter 5, “Cache Model and Memory Coherency,” in the *Programming Environments Manual*.

2.3.4.3.2 Integer Load and Store Address Generation

Integer load and store operations generate effective addresses using register indirect with immediate index mode, register indirect with index mode, or register indirect mode. See [Section 2.3.2.3, “Effective Address Calculation,”](#) for information about calculating effective addresses. Note that in some implementations, operations that are not naturally aligned can suffer performance degradation. Refer to [Section 4.6.6, “Alignment Interrupt \(0x00600\),”](#) for additional information about load and store address alignment interrupts.

2.3.4.3.3 Register Indirect Integer Load Instructions

For integer load instructions, the byte, half word, word, or double word addressed by the EA (effective address) is loaded into **rD**. Many integer load instructions have an update form, in which **rA** is updated with the generated effective address. For these forms, if **rA** ≠ 0 and **rA** ≠ **rD** (otherwise invalid), the EA

is placed into **rA** and the memory element (byte, half word, word, or double word) addressed by the EA is loaded into **rD**. Note that the PowerPC architecture defines load with update instructions with operand **rA = 0** or **rA = rD** as invalid forms.

Implementation Notes—The following notes describe the e600 core implementation of integer load instructions:

- The PowerPC architecture cautions programmers that some implementations of the architecture can execute the load half algebraic (**lha**, **lhax**) instructions with greater latency than other types of load instructions. This is not the case for the e600 core; these instructions operate with the same latency as other load instructions.
- The PowerPC architecture cautions programmers that some implementations of the architecture can run the load/store byte-reverse (**lhbrx**, **lbrx**, **sthbrx**, **stwbrx**) instructions with greater latency than other types of load/store instructions. This is not the case for the e600 core. These instructions operate with the same latency as the other load/store instructions.
- The PowerPC architecture describes some preferred instruction forms for load and store multiple instructions and integer move assist instructions that can perform better than other forms in some implementations. None of these preferred forms affect instruction performance on the e600 core. Usage of load/store string instruction is discouraged.
- The PowerPC architecture defines the **lwarx** and **stwcx** as a way to update memory atomically. In the e600 core, reservations are made on behalf of aligned 32-byte sections of the memory address space. Executing **lwarx** and **stwcx** to a page marked write-through does cause a DSI interrupt if the page is marked cacheable write-through (WIM = 10x) or caching-inhibited (WIM = x1x), but as with other memory accesses, DSI interrupts can result for other reasons such as a protection violations or page faults.

Table 2-49 summarizes the integer load instructions.

Table 2-49. Integer Load Instructions

Name	Mnemonic	Syntax
Load Byte and Zero	lbz	rD,d(rA)
Load Byte and Zero Indexed	lbzx	rD,rA,rB
Load Byte and Zero with Update	lbzu	rD,d(rA)
Load Byte and Zero with Update Indexed	lbzux	rD,rA,rB
Load Half Word and Zero	lhz	rD,d(rA)
Load Half Word and Zero Indexed	lhzx	rD,rA,rB
Load Half Word and Zero with Update	lhzu	rD,d(rA)
Load Half Word and Zero with Update Indexed	lhzux	rD,rA,rB
Load Half Word Algebraic	lha	rD,d(rA)
Load Half Word Algebraic Indexed	lhax	rD,rA,rB
Load Half Word Algebraic with Update	lhau	rD,d(rA)
Load Half Word Algebraic with Update Indexed	lhaux	rD,rA,rB

Table 2-49. Integer Load Instructions (continued)

Name	Mnemonic	Syntax
Load Word and Zero	lwz	rD,d(rA)
Load Word and Zero Indexed	lwzx	rD,rA,rB
Load Word and Zero with Update	lwzu	rD,d(rA)
Load Word and Zero with Update Indexed	lwzux	rD,rA,rB

2.3.4.3.4 Integer Store Instructions

For integer store instructions, the contents of **rS** are stored into the byte, half word, word or double word in memory addressed by the EA (effective address). Many store instructions have an update form, in which **rA** is updated with the EA. For these forms, the following rules apply:

- If **rA** \neq 0, the effective address is placed into **rA**.
- If **rS** = **rA**, the contents of register **rS** are copied to the target memory element, then the generated EA is placed into **rA** (**rS**).

The PowerPC architecture defines store with update instructions with **rA** = 0 as an invalid form. In addition, it defines integer store instructions with the CR update option enabled (Rc field, bit 31, in the instruction encoding = 1) to be an invalid form. [Table 2-50](#) summarizes the integer store instructions.

Table 2-50. Integer Store Instructions

Name	Mnemonic	Syntax
Store Byte	stb	rS,d(rA)
Store Byte Indexed	stbx	rS,rA,rB
Store Byte with Update	stbu	rS,d(rA)
Store Byte with Update Indexed	stbux	rS,rA,rB
Store Half Word	sth	rS,d(rA)
Store Half Word Indexed	sthx	rS,rA,rB
Store Half Word with Update	sthu	rS,d(rA)
Store Half Word with Update Indexed	sthux	rS,rA,rB
Store Word	stw	rS,d(rA)
Store Word Indexed	stwx	rS,rA,rB
Store Word with Update	stwu	rS,d(rA)
Store Word with Update Indexed	stwux	rS,rA,rB

2.3.4.3.5 Integer Store Gathering

The e600 core performs store gathering for write-through accesses to nonguarded space or to cache-inhibited stores to nonguarded space if the requirements described in [Section 3.1.2.3, “Store Gathering/Merging,”](#) are met. These stores are combined in the load/store unit (LSU) to form a double word or quad word and are sent out on the MPX bus as a single operation. However, stores can be gathered

only if the successive stores that meet the criteria are queued and pending. The e600 core also performs store merging as described in [Section 3.1.2.3, “Store Gathering/Merging,”](#)

Store gathering takes place regardless of the address order of the stores. The store gathering and merging feature is enabled by setting `HID0[SGE]`.

If store gathering is enabled and the stores do not fall under the above categories, an `eieio` or `sync` instruction must be used to prevent two stores from being gathered.

2.3.4.3.6 Integer Load and Store with Byte-Reverse Instructions

[Table 2-51](#) describes integer load and store with byte-reverse instructions. When used in a system operating with the default big-endian byte order, these instructions have the effect of loading and storing data in little-endian order. Likewise, when used in a system operating with little-endian byte order, these instructions have the effect of loading and storing data in big-endian order. For more information about big-endian and little-endian byte ordering, see “Byte Ordering,” in Chapter 3, “Operand Conventions,” in the *Programming Environments Manual*.

Table 2-51. Integer Load and Store with Byte-Reverse Instructions

Name	Mnemonic	Syntax
Load Half Word Byte-Reverse Indexed	<code>lhbrx</code>	<code>rD,rA,rB</code>
Load Word Byte-Reverse Indexed	<code>lwbrx</code>	<code>rD,rA,rB</code>
Store Half Word Byte-Reverse Indexed	<code>sthbrx</code>	<code>rS,rA,rB</code>
Store Word Byte-Reverse Indexed	<code>stwbrx</code>	<code>rS,rA,rB</code>

2.3.4.3.7 Integer Load and Store Multiple Instructions

The load/store multiple instructions are used to move blocks of data to and from the GPRs. The load multiple and store multiple instructions can have operands that require memory accesses crossing a 4-Kbyte page boundary. As a result, these instructions can be interrupted by a DSI interrupt associated with the address translation of the second page.

The PowerPC architecture defines the Load Multiple Word (`lmw`) instruction with `rA` in the range of registers to be loaded as an invalid form.

Table 2-52. Integer Load and Store Multiple Instructions

Name	Mnemonic	Syntax
Load Multiple Word	<code>lmw</code>	<code>rD,d(rA)</code>
Store Multiple Word	<code>stmw</code>	<code>rS,d(rA)</code>

2.3.4.3.8 Integer Load and Store String Instructions

The integer load and store string instructions allow movement of data from memory to registers or from registers to memory without concern for alignment. These instructions can be used for a short move between arbitrary memory locations or to initiate a long move between misaligned memory fields. However, in some implementations, these instructions are likely to have greater latency and take longer to

execute, perhaps much longer, than a sequence of individual load or store instructions that produce the same results. [Table 2-53](#) summarizes the integer load and store string instructions.

Table 2-53. Integer Load and Store String Instructions

Name	Mnemonic	Syntax
Load String Word Immediate	lswi	rD,rA,NB
Load String Word Indexed	lswx	rD,rA,rB
Store String Word Immediate	stswi	rS,rA,NB
Store String Word Indexed	stswx	rS,rA,rB

In the e600 core implementation, operating with little-endian byte order, execution of a load or string instruction will take an alignment interrupt.

Load string and store string instructions can involve operands that are not word-aligned.

For load/store string operations, the e600 core does not combine register values to reduce the number of discrete accesses. However, if store gathering is enabled and the accesses fall under the criteria for store gathering the stores can be combined to enhance performance. At a minimum, additional cache access cycles are required. Usage of load/store string instructions is discouraged.

2.3.4.3.9 Floating-Point Load and Store Address Generation

Floating-point load and store operations generate effective addresses using the register indirect with immediate index addressing mode and register indirect with index addressing mode. Floating-point loads and stores are not supported for direct-store accesses. The use of floating-point loads and stores for direct-store access results in an alignment interrupt.

There are two forms of the floating-point load instruction—single-precision and double-precision operand formats. Because the FPRs support only the floating-point double-precision format, single-precision floating-point load instructions convert single-precision data to double-precision format before loading an operand into an FPR.

Implementation Note—The e600 core treats interrupts as follows:

- The FPU operates in either ignore exceptions mode ($MSR[FE0] = MSR[FE1] = 0$) or precise mode (any other settings for $MSR[FE0,FE1]$). For the e600 core, ignore exceptions mode allows floating-point instructions to complete earlier and thus can provide better performance than precise mode.

The floating-point load and store indexed instructions (**lfsx**, **lfsux**, **lfdx**, **lfdux**, **stfsx**, **stfsux**, **stfdx**, **stfdux**) are invalid when the Rc bit is one. The PowerPC architecture defines a load with update instruction with $rA = 0$ as an invalid form. [Table 2-54](#) summarizes the floating-point load instructions.

Table 2-54. Floating-Point Load Instructions

Name	Mnemonic	Syntax
Load Floating-Point Single	lfs	frD,d(rA)
Load Floating-Point Single Indexed	lfsx	frD,rA,rB

Table 2-54. Floating-Point Load Instructions (continued)

Name	Mnemonic	Syntax
Load Floating-Point Single with Update	lfsu	frD,d(rA)
Load Floating-Point Single with Update Indexed	lfsux	frD,rA,rB
Load Floating-Point Double	lfd	frD,d(rA)
Load Floating-Point Double Indexed	lfdx	frD,rA,rB
Load Floating-Point Double with Update	lfdu	frD,d(rA)
Load Floating-Point Double with Update Indexed	lfdux	frD,rA,rB

2.3.4.3.10 Floating-Point Store Instructions

This section describes floating-point store instructions. There are three basic forms of the store instruction—single-precision, double-precision, and integer. The integer form is supported by the optional **stfiwx** instruction. Because the FPRs support only floating-point, double-precision format for floating-point data, single-precision floating-point store instructions convert double-precision data to single-precision format before storing the operands. [Table 2-55](#) summarizes the floating-point store instructions.

Table 2-55. Floating-Point Store Instructions

Name	Mnemonic	Syntax
Store Floating-Point Single	stfs	frS,d(rA)
Store Floating-Point Single Indexed	stfsx	frS,r B
Store Floating-Point Single with Update	stfsu	frS,d(rA)
Store Floating-Point Single with Update Indexed	stfsux	frS,r B
Store Floating-Point Double	stfd	frS,d(rA)
Store Floating-Point Double Indexed	stfdx	frS,rB
Store Floating-Point Double with Update	stfdu	frS,d(rA)
Store Floating-Point Double with Update Indexed	stfdux	frS,r B
Store Floating-Point as Integer Word Indexed ¹	stfiwx	frS,rB

¹ The **stfiwx** instruction is optional to the PowerPC architecture

Some floating-point store instructions require conversions in the LSU. [Table 2-56](#) shows conversions the LSU makes when executing a Store Floating-Point Single instruction.

Table 2-56. Store Floating-Point Single Behavior

FPR Precision	Data Type	Action
Single	Normalized	Store
Single	Denormalized	Store
Single	Zero, infinity, QNaN	Store

Table 2-56. Store Floating-Point Single Behavior (continued)

FPR Precision	Data Type	Action
Single	SNaN	Store
Double	Normalized	If ($\text{exp} \leq 896$) then Denormalize and Store else Store
Double	Denormalized	Store zero
Double	Zero, infinity, QNaN	Store
Double	SNaN	Store

Table 2-57 shows the conversions made when performing a Store Floating-Point Double instruction. Most entries in the table indicate that the floating-point value is simply stored. Only in a few cases are any other actions taken.

Table 2-57. Store Floating-Point Double Behavior

FPR Precision	Data Type	Action
Single	Normalized	Store
Single	Denormalized	Normalize and Store
Single	Zero, infinity, QNaN	Store
Single	SNaN	Store
Double	Normalized	Store
Double	Denormalized	Store
Double	Zero, infinity, QNaN	Store
Double	SNaN	Store

Architecturally, all floating-point numbers are represented in double-precision format within the e600 core. Execution of a store floating-point single (**stfs**, **stfsu**, **stfsx**, **stfsux**) instruction requires conversion from double- to single-precision format. If the exponent is not greater than 896, this conversion requires denormalization. The e600 core supports this denormalization by shifting the mantissa one bit at a time. Anywhere from 1 to 23 clock cycles are required to complete the denormalization, depending upon the value to be stored.

Because of how floating-point numbers are implemented in the e600 core, there is also a case when execution of a store floating-point double (**stfd**, **stfdu**, **stfdx**, **stfdux**) instruction can require internal shifting of the mantissa. This case occurs when the operand of a store floating-point double instruction is a denormalized single-precision value. The value could be the result of a load floating-point single instruction, a single-precision arithmetic instruction, or a floating round to single-precision instruction. In these cases, shifting the mantissa takes from 1 to 23 clock cycles, depending upon the value to be stored. These cycles are incurred during the store.

2.3.4.4 Branch and Flow Control Instructions

Some branch instructions can redirect instruction execution conditionally based on the value of bits in the CR. When the processor encounters one of these instructions, it scans the execution pipelines to determine whether an instruction in progress can affect the particular CR bit. If no interlock is found, the branch can be resolved immediately by checking the bit in the CR and taking the action defined for the branch instruction.

2.3.4.4.1 Branch Instruction Address Calculation

Branch instructions can alter the sequence of instruction execution. Instruction addresses are always assumed to be word aligned; the processors that ignore the two low-order bits of the generated branch target address.

Branch instructions compute the EA of the next instruction address using the following addressing modes:

- Branch relative
- Branch conditional to relative address
- Branch to absolute address
- Branch conditional to absolute address
- Branch conditional to link register
- Branch conditional to count register

Note that in the e600 core, all branch instructions (**b**, **ba**, **bl**, **bla**, **bc**, **bca**, **bcl**, **bcla**, **bclr**, **bclrl**, **bcctr**, **bcctrl**) are executed in the BPU and condition register logical instructions (**crand**, **cror**, **crxor**, **crnand**, **crnor**, **crandc**, **creqv**, **crorc**, and **mcrf**) are executed by the IU2. Some of these instructions can redirect instruction execution conditionally on the value of CR, CTR, or LR bits. When the CR bits resolve, the branch instruction is either marked as correct or mispredicted. Correcting a mispredicted branch requires that the e600 core flush speculatively executed instructions and restore the machine state to immediately after the branch. This correction can be done when all non-speculative instructions older than the mispredicting branch have completed.

2.3.4.4.2 Branch Instructions

Table 2-58 lists the branch instructions provided by the processors that implement the PowerPC architecture. To simplify assembly language programming, a set of simplified mnemonics and symbols is provided for the most frequently used forms of branch conditional, compare, trap, rotate and shift, and certain other instructions. See Appendix F, “Simplified Mnemonics,” in the *Programming Environments Manual* for a list of simplified mnemonic examples.

Table 2-58. Branch Instructions

Name	Mnemonic	Syntax
Branch	b (ba bl bla)	target_addr
Branch Conditional	bc (bca bcl bcla)	BO,BI,target_addr
Branch Conditional to Link Register	bclr (bclrl)	BO,BI
Branch Conditional to Count Register	bcctr (bcctrl)	BO,BI

2.3.4.4.3 Condition Register Logical Instructions

Condition register logical instructions, shown in [Table 2-59](#), and the Move Condition Register Field (**mcrf**) instruction are also defined as flow control instructions.

Table 2-59. Condition Register Logical Instructions

Name	Mnemonic	Syntax
Condition Register AND	crand	crbD,crbA,crbB
Condition Register OR	cror	crbD,crbA,crbB
Condition Register XOR	crxor	crbD,crbA,crbB
Condition Register NAND	crnand	crbD,crbA,crbB
Condition Register NOR	crnor	crbD,crbA,crbB
Condition Register Equivalent	creqv	crbD,crbA, crbB
Condition Register AND with Complement	crandc	crbD,crbA, crbB
Condition Register OR with Complement	crorc	crbD,crbA, crbB
Move Condition Register Field	mcrf	crfD,crfS

Note that if the LR update option is enabled for any of these instructions, the PowerPC architecture defines these forms of the instructions as invalid.

2.3.4.4.4 Trap Instructions

The trap instructions shown in [Table 2-60](#) are provided to test for a specified set of conditions. If any of the conditions tested by a trap instruction are met, the system trap type program interrupt is taken. For more information, see [Section 4.6.7, “Program Interrupt \(0x00700\).”](#) If the tested conditions are not met, instruction execution continues normally.

Table 2-60. Trap Instructions

Name	Mnemonic	Syntax
Trap Word Immediate	twi	TO,rA,SIMM
Trap Word	tw	TO,rA,rB

See Appendix F, “Simplified Mnemonics,” in the *Programming Environments Manual* for a complete set of simplified mnemonics.

2.3.4.5 System Linkage Instruction—UISA

The System Call (**sc**) instruction permits a program to call on the system to perform a service; see [Table 2-61](#) and also [Section 2.3.6.1, “System Linkage Instructions—OEA,”](#) for additional information.

Table 2-61. System Linkage Instruction—UISA

Name	Mnemonic	Syntax
System Call	sc	—

Executing this instruction causes the system call interrupt handler to be evoked. For more information, see [Section 4.6.10, “System Call Interrupt \(0x00C00\).”](#)

2.3.4.6 Processor Control Instructions—UISA

Processor control instructions are used to read from and write to the condition register (CR), machine state register (MSR), and special-purpose registers (SPRs). See [Section 2.3.5.1, “Processor Control Instructions—VEA,”](#) for the **mftb** instruction and [Section 2.3.6.2, “Processor Control Instructions—OEA,”](#) for information about the instructions used for reading from and writing to the MSR and SPRs.

2.3.4.6.1 Move To/From Condition Register Instructions

[Table 2-62](#) summarizes the instructions for reading from or writing to the condition register.

Table 2-62. Move To/From Condition Register Instructions

Name	Mnemonic	Syntax
Move to Condition Register Fields	mtrcf	CRM,rS
Move to Condition Register from XER	mcrxr	crfD
Move from Condition Register	mfcrr	rD

Implementation Note—The PowerPC architecture indicates that in some implementations the Move to Condition Register Fields (**mtrcf**) instruction can perform more slowly when only a portion of the fields are updated as opposed to all of the fields. The condition register access latency for the e600 core is the same in both cases, if multiple fields are affected. Note that an **mtrcf** to a single field is handled in the IU1s and latency may be lower if an **mtrcf** multi is split into its component single field pieces by the compiler.

2.3.4.6.2 Move To/From Special-Purpose Register Instructions (UISA)

[Table 2-63](#) lists the **mtspr** and **mfspr** instructions.

Table 2-63. Move To/From Special-Purpose Register Instructions (UISA)

Name	Mnemonic	Syntax
Move to Special-Purpose Register	mtspr	SPR,rS
Move from Special-Purpose Register	mfspr	rD,SPR

[Table 2-64](#) lists the SPR numbers for user-level PowerPC SPR accesses.

Encodings for the e600-specific user-level SPRs are listed in [Table 2-65](#).

Table 2-64. User-Level PowerPC SPR Encodings

Register Name	SPR ¹			Access	mfspr/mtspr
	Decimal	spr[5–9]	spr[0–4]		
CTR	9	00000	01001	User (UISA)	Both
LR	8	00000	01000	User (UISA)	Both
TBL ²	268	01000	01100	User (VEA)	mftb
TBU ²	269	01000	01101	User (VEA)	mftb
VRSAVE ³	256	01000	00000	User (AltiVec/UISA)	Both
XER	1	00000	00001	User (UISA)	Both

¹ Note that the order of the two 5-bit halves of the SPR number is reversed compared with actual instruction coding. For **mtspr** and **mfspr** instructions, the SPR number coded in assembly language does not appear directly as a 10-bit binary number in the instruction. The number coded is split into two 5-bit halves that are reversed in the instruction, with the high-order 5 bits appearing in bits 16–20 of the instruction and the low-order 5 bits in bits 11–15.

² The TB registers are referred to as TBRs rather than SPRs and can be written to using the **mtspr** instruction in supervisor mode and the TBR numbers here. The TB registers can be read in user mode using either the **mftb** instruction and specifying TBR 268 for TBL and TBR 269 for TBU.

³ Register defined by the AltiVec technology.

Table 2-65. User-Level SPR Encodings for e600-Defined Registers

Register Name	SPR ¹			Access	mfspr/mtspr
	Decimal	spr[5–9]	spr[0–4]		
UMMCR0	936	11101	01000	User	mfspr
UMMCR1	940	11101	01100	User	mfspr
UMMCR2	928	11101	00000	User	mfspr
UPMC1	937	11101	01001	User	mfspr
UPMC2	938	11101	01010	User	mfspr
UPMC3	941	11101	01101	User	mfspr
UPMC4	942	11101	01110	User	mfspr
UPMC5	929	11101	00001	User	mfspr
UPMC6	930	11101	00010	User	mfspr
USIAR	939	11101	01011	User	mfspr

¹ Note that the order of the two 5-bit halves of the SPR number is reversed compared with actual instruction coding. For **mtspr** and **mfspr** instructions, the SPR number coded in assembly language does not appear directly as a 10-bit binary number in the instruction. The number coded is split into two 5-bit halves that are reversed in the instruction, with the high-order 5 bits appearing in bits 16–20 of the instruction and the low-order 5 bits in bits 11–15.

2.3.4.7 Memory Synchronization Instructions—UISA

Memory synchronization instructions control the order in which memory operations are completed with respect to asynchronous events, and the order in which memory operations are seen by other processors or memory access mechanisms. See [Section 3.3.3.6, “Atomic Memory References,”](#) for additional information about these instructions and about related aspects of memory synchronization. See [Table 2-66](#) for a summary.

Table 2-66. Memory Synchronization Instructions—UISA

Name	Mnemonic	Syntax	Implementation Notes
Load Word and Reserve Indexed	lwarx ¹	rD,rA,rB	Programmers can use lwarx with stwcx. to emulate common semaphore operations such as test and set, compare and swap, exchange memory, and fetch and add. Both instructions must use the same EA. Reservation granularity is implementation-dependent. The e600 core makes reservations on behalf of aligned 32-byte sections of the memory address space. Executing lwarx and stwcx. to a page marked write-through (WIMG = 10xx) or caching-inhibited (WIMG = x1xx) or when the data cache is disabled or locked causes a DSI interrupt. If the location is not word-aligned, an alignment interrupt occurs. The stwcx. instruction is the only load/store instruction with a valid form if Rc is set. If Rc is zero, executing stwcx. sets CR0 to an undefined value.
Store Word Conditional Indexed	stwcx. ¹	rS,rA,rB	
Synchronize	sync	—	Because it delays execution of subsequent instructions until all previous instructions complete to where they cannot cause an interrupt, sync is a barrier against store gathering. Additionally, all load/store cache/MPX bus activities initiated by prior instructions are completed. Touch load operations (dcbt , dcbtst) must complete address translation, but need not complete on the MPX bus. The sync completes after a successful broadcast on the MPX bus. The latency of sync depends on the processor state when it is dispatched and on various system-level situations. Note that frequent use of sync will degrade performance.

¹ Note that the e600 core implements **lwarx** and **stwcx.** as defined in the PowerPC architecture version 1.10. The execution of an **lwarx** or **stwcx.** instruction to memory marked write-through or cache-inhibited causes a DSI interrupt.

Integrated devices with additional caches should take special care to recognize the hardware signaling caused by a SYNC MPX bus operation and perform the appropriate actions to guarantee that memory references that can be queued internally to the additional cache have been performed globally.

See [Section 2.3.5.2, “Memory Synchronization Instructions—VEA,”](#) for details about additional memory synchronization (**eieio**) instructions.

In the PowerPC architecture, the Rc bit must be zero for most load and store instructions. If Rc is set, the instruction form is invalid for **sync** and **lwarx** instructions. If the e600 core encounters one of these invalid instruction forms, it sets CR0 to an undefined value.

2.3.5 PowerPC VEA Instructions

The PowerPC virtual environment architecture (VEA) describes the semantics of the memory model that can be assumed by software processes, and includes descriptions of the cache model, cache control instructions, address aliasing, and other related issues. Implementations that conform to the VEA also adhere to the UISA, but do not necessarily adhere to the OEA.

This section describes additional instructions that are provided by the VEA.

2.3.5.1 Processor Control Instructions—VEA

In addition to the move to condition register instructions (specified by the UISA), the VEA defines the **mftb** instruction (user-level instruction) for reading the contents of the time base register; see [Chapter 3, “L1 and L2 Cache Operation,”](#) for more information. [Table 2-67](#) shows the **mftb** instruction.

Table 2-67. Move From Time Base Instruction

Name	Mnemonic	Syntax
Move from Time Base	mftb	rD, TBR

Simplified mnemonics are provided for the **mftb** instruction so it can be coded with the TBR name as part of the mnemonic rather than requiring it to be coded as an operand. See Appendix F, “Simplified Mnemonics,” in the *Programming Environments Manual* for simplified mnemonic examples and for simplified mnemonics for Move from Time Base (**mftb**) and Move from Time Base Upper (**mftbu**), which are variants of the **mftb** instruction rather than of **mfspr**. The **mftb** instruction serves as both a basic and simplified mnemonic. Assemblers recognize an **mftb** mnemonic with two operands as the basic form, and an **mftb** mnemonic with one operand as the simplified form.

Implementation Note—In the e600 core, note the following:

- The e600 core allows user-mode read access to the time base counter through the use of the Move from Time Base (**mftb**) instruction. As a 32-bit implementation of the PowerPC architecture, the e600 core can access TBU and TBL separately only.
- The time base counter is clocked at a frequency that is one-fourth that of the bus clock. Counting is enabled by assertion of the time base enable (*tben*) input signal.

2.3.5.2 Memory Synchronization Instructions—VEA

Memory synchronization instructions control the order in which memory operations are completed with respect to asynchronous events, and the order in which memory operations are seen by other processors or memory access mechanisms. See [Chapter 3, “L1 and L2 Cache Operation,”](#) for more information about these instructions and about related aspects of memory synchronization.

In addition to the **sync** instruction (specified by UISA), the VEA defines the Enforce In-Order Execution of I/O (**eiio**) and Instruction Synchronize (**isync**) instructions. The number of cycles required to complete an **eiio** instruction depends on system parameters and on the core’s state when the instruction is issued. As a result, frequent use of this instruction can degrade performance. Note that the broadcast of these instructions on the MPX bus is controlled by the HID1[SYNCBE] bit.

Table 2-68 describes the memory synchronization instructions defined by the VEA.

Table 2-68. Memory Synchronization Instructions—VEA

Name	Mnemonic	Syntax	Implementation Notes
Enforce In-Order Execution of I/O	eieio	—	The eieio instruction is dispatched to the LSU and executes after all previous cache-inhibited or write-through accesses are performed; all subsequent instructions that generate such accesses execute after eieio . As the eieio operation does not affect the caches, it bypasses the L2 cache and is forwarded to the MPX bus. An EIEIO operation is broadcast on the MPX bus to enforce ordering in the memory system external to the core. Because the e600 core does reorder noncacheable accesses, eieio may be needed to force ordering. However, if store gathering is enabled and an eieio is detected in a store queue, stores are not gathered. Broadcasting eieio prevents other peripherals, such as bus bridge chips, from gathering stores.
Instruction Synchronize	isync	—	The isync instruction is refetch serializing; that is, it causes the e600 core to wait for all prior instructions to complete first then executes, purging all instructions from the core and then refetching the next instruction. The isync instruction is not executed until all previous instructions complete to the point where they cannot cause an interrupt. The isync instruction does not wait for all pending stores in the store queue to complete. Any instruction after an isync sees all effects of prior instructions occurring before the isync .

2.3.5.3 Memory Control Instructions—VEA

Memory control instructions can be classified as follows:

- Cache management instructions (user-level and supervisor-level)
- Translation lookaside buffer management instructions (OEA)

This section describes the user-level cache management instructions defined by the VEA. See [Section 2.3.6.3, “Memory Control Instructions—OEA,”](#) for information about supervisor-level cache, segment register manipulation, and translation lookaside buffer management instructions. For a complete description of the MPX bus operations caused by cache control instructions, see [Section 3.7.1, “MPX bus Operations Caused by Cache Control Instructions.”](#)

2.3.5.3.1 User-Level Cache Instructions—VEA

The instructions summarized in this section help user-level programs manage caches within the core if they are implemented. See [Chapter 3, “L1 and L2 Cache Operation,”](#) for more information about cache topics. The following sections describe how these operations are treated with respect to the e600 core’s caches.

As with other memory-related instructions, the effects of cache management instructions on memory are weakly-ordered. If the programmer must ensure that cache or other instructions have been performed with respect to all other processors and system mechanisms, a **sync** instruction must be placed after those instructions.

Note that the e600 core interprets cache control instructions (**icbi**, **dcbi**, **dcbf**, **dcbz**, and **dcbst**) as if they pertain only to the local L1 and L2 caches. A **dcbz** (with M set) is always broadcast on the core interface if it does not hit as modified in any core cache.

All cache control instructions to direct-store space function as no-ops. For information on how cache control instructions affect the L2 cache, see [3.6.4, “L2 Cache Operation.”](#)

Section 3.6.3.1.5, “Flushing of L1 and L2 Caches,” Table 2-69 summarizes the cache instructions defined by the VEA. Note that these instructions are accessible to user-level programs.

Table 2-69. User-Level Cache Instructions

Name	Mnemonic	Syntax	Implementation Notes
Data Cache Block Touch ¹	dcbt	rA,rB	<p>The VEA defines this instruction to allow for potential system performance enhancements through the use of software-initiated prefetch hints. Implementations are not required to take any action based on execution of this instruction, but they can prefetch the cache block corresponding to the EA into their cache. When dcbt executes, the e600 core checks for protection violations (as for a load instruction). This instruction is treated as a no-op for the following cases:</p> <ul style="list-style-type: none"> • The access causes a protection violation. • The page is mapped cache-inhibited or direct-store (T = 1). • The cache is locked or disabled • HID0[NOPTI] = 1 <p>Otherwise, if no data is in the cache location, the e600 core requests a cache line fill. Data brought into the cache is validated as if it were a load instruction. The memory reference of a dcbt sets the referenced bit.</p>
Data Cache Block Touch for Store ¹	dcbtst	rA,rB	<p>This instruction dcbtst can be no-oped by setting HID0[NOPTI]. The dcbtst instruction behaves similarly to a dcbt instruction, except that the line fill request on the MPX bus is signaled as read or read-claim, and the data is marked as exclusive in the L1 data cache if there is no shared response on the MPX bus. More specifically, the following cases occur depending on where the line currently exists or does not exist in the e600 core.</p> <ul style="list-style-type: none"> • dcbtst hits in the L1 data cache. In this case, the dcbtst does nothing and the state of the line in the cache is not changed. Thus, if the line was in the shared state, a subsequent store hits on this shared line and incurs the associated latency penalties. • dcbtst misses in the L1 data cache and hits in the L2 cache. In this case, the dcbtst will reload the L1 data cache with the state found in the L2 cache. Again, if the line was in the shared state in the L2, a subsequent store will hit on this shared line and incur the associated latency penalties. • dcbtst misses in L1 data cache and L2 cache. In this case, the e600 core will request the line from memory with read or read-claim and reload the L1 data cache in the exclusive state. As subsequent store will hit on exclusive and can perform the store to the L1 data cache immediately. <p>In addition, a dcbtst instruction will be no-oped if the target address of the dcbtst is mapped as write-through.</p>
Data Cache Block Set to Zero	dcbz	rA,rB	<p>The EA is computed, translated, and checked for protection violations. For cache hits, 32 bytes of zeros are written to the cache block and the tag is marked modified. For cache misses with the replacement block marked not modified, the zero reload is performed and the cache block is marked modified. However, if the replacement block is marked modified, the contents are written back to memory first. The instruction takes an alignment interrupt if the cache block is locked or disabled or if the cache is marked WT or CI. If WIMG = xx1x (coherency enforced), the address is broadcast to the MPX bus before the zero reload fill. The interrupt priorities (from highest to lowest) are as follows:</p> <ol style="list-style-type: none"> 1 Cache disabled—Alignment interrupt 2 Cache is locked—Alignment interrupt 3 Page marked write-through or cache-inhibited—alignment interrupt 4 BAT protection violation—DSI interrupt 5 TLB protection violation—DSI interrupt <p>dcbz is broadcast if WIMG = xx1x (coherency enforced).</p>

Table 2-69. User-Level Cache Instructions (continued)

Name	Mnemonic	Syntax	Implementation Notes
Data Cache Block Allocate	dcba	rA,rB	<p>The EA is computed, translated, and checked for protection violations. For cache hits, 32 bytes of zeros are written to the cache block and the tag is marked modified. For cache misses with the replacement block marked non-dirty, the zero reload is performed and the cache block is marked modified. However, if the replacement block is marked modified, the contents are written back to memory first. The instruction performs a no-op if the cache is locked or disabled or if the cache is marked WT or CI. If WIMG = xx1x (coherency enforced), the address is broadcast to the MPX bus before the zero reload fill.</p> <p>A no-op occurs for the following:</p> <ul style="list-style-type: none"> • Cache is disabled • Cache is locked • Page marked write-through or cache-inhibited • BAT protection violation • TLB protection violation <p>dcba is broadcast if WIMG = xx1x (coherency enforced).</p>
Data Cache Block Store	dcbst	rA,rB	<p>The EA is computed, translated, and checked for protection violations.</p> <ul style="list-style-type: none"> • For cache hits with the tag marked not modified, no further action is taken. • For cache hits with the tag marked modified, the cache block is written back to memory and marked exclusive. <p>If WIMG = xx1x (coherency enforced), dcbst is broadcast. The instruction acts like a load with respect to address translation and memory protection. It executes regardless of whether the cache is disabled or locked.</p> <p>The interrupt priorities (from highest to lowest) for dcbst are as follows:</p> <ol style="list-style-type: none"> 1 BAT protection violation—DSI interrupt 2 TLB protection violation—DSI interrupt
Data Cache Block Flush	dcbf	rA,rB	<p>The EA is computed, translated, and checked for protection violations:</p> <ul style="list-style-type: none"> • For cache hits with the tag marked modified, the cache block is written back to memory and the cache entry is invalidated. • For cache hits with the tag marked not modified, the entry is invalidated. • For cache misses, no further action is taken. <p>A dcbf is broadcast if WIMG = xx1x (coherency enforced). The instruction acts like a load with respect to address translation and memory protection. It executes regardless of whether the cache is disabled or locked.</p> <p>The interrupt priorities (from highest to lowest) for dcbf are as follows:</p> <ol style="list-style-type: none"> 1 BAT protection violation—DSI interrupt 2 TLB protection violation—DSI interrupt
Instruction Cache Block Invalidate	icbi	rA,rB	<p>This instruction is broadcast on the MPX bus if WIMG = xx1x. icbi should always be followed by a sync and an isync to make sure that the effects of the icbi are seen by the instruction fetches following the icbi itself.</p>

¹ A program that uses **dcbt** and **dcbtst** instructions improperly performs less efficiently. To improve performance, HID0[NOPTI] can be set, which causes **dcbt** and **dcbtst** to be no-oped at the cache. They do not cause MPX bus activity and cause only a 1-clock execution latency. The default state of this bit is zero which enables the use of these instructions.

2.3.5.4 Optional External Control Instructions

The PowerPC architecture defines an optional external control feature that, if implemented, is supported by the two external control instructions, **eciwx** and **ecowx**. These instructions are not implemented on the e600 core. Note that the EAR, which is accessible only in supervisor mode, must not be enabled. Otherwise, attempted execution of **eciwx/ecowx** causes boundedly undefined results.

2.3.6 PowerPC OEA Instructions

The PowerPC operating environment architecture (OEA) includes the structure of the memory management model, supervisor-level registers, and the interrupt model. Implementations that conform to the OEA also adhere to the UISA and the VEA. This section describes the instructions provided by the OEA.

2.3.6.1 System Linkage Instructions—OEA

This section describes the system linkage instructions (see [Table 2-70](#)). The user-level **sc** instruction lets a user program call on the system to perform a service and causes the processor to take a system call interrupt. The supervisor-level **rfi** instruction is used for returning from an interrupt handler.

Table 2-70. System Linkage Instructions—OEA

Name	Mnemonic	Syntax	Implementation Notes
System Call	sc	—	The sc instruction is context-synchronizing.
Return from Interrupt	rfi	—	rfi is context-synchronizing. For the e600 core, this means that rfi works its way to the final stage of the execution pipeline, updates architected registers, and redirects the instruction flow.

2.3.6.2 Processor Control Instructions—OEA

The instructions listed in [Table 2-71](#) provide access to the segment registers for 32-bit implementations. These instructions operate completely independently of the MSR[IR] and MSR[DR] bit settings. Refer to “Synchronization Requirements for Special Registers and for Lookaside Buffers,” in Chapter 2, “Register Set,” of the *Programming Environments Manual* for serialization requirements and other recommended precautions to observe when manipulating the segment registers.

Table 2-71. Segment Register Manipulation Instructions (OEA)

Name	Mnemonic	Syntax	Implementation Notes
Move to Segment Register	mtsr	SR,rS	—
Move to Segment Register Indirect	mtsrin	rS,rB	—
Move from Segment Register	mfsr	rD,SR	—
Move from Segment Register Indirect	mfsrin	rD,rB	—

The processor control instructions used to access the MSR and the SPRs are discussed in this section. [Table 2-72](#) lists instructions for accessing the MSR.

Table 2-72. Move To/From Machine State Register Instructions

Name	Mnemonic	Syntax
Move to Machine State Register	mtmsr	rS
Move from Machine State Register	mfmsr	rD

The OEA defines encodings of **mtspr** and **mfspir** to provide access to supervisor-level registers. The instructions are listed in [Table 2-73](#).

Table 2-73. Move To/From Special-Purpose Register Instructions (OEA)

Name	Mnemonic	Syntax
Move to Special-Purpose Register	mtspr	SPR,rS
Move from Special-Purpose Register	mfspir	rD,SPR

Encodings for the architecture-defined SPRs are listed in [Table 2-64](#). Encodings for e600-specific, supervisor-level SPRs are listed in [Table 2-65](#). Simplified mnemonics are provided for **mtspr** and **mfspir** in Appendix F, “Simplified Mnemonics,” in the *Programming Environments Manual*.

[Table 2-74](#) lists the SPR numbers for supervisor-level PowerPC SPR accesses.

Table 2-74. Supervisor-Level PowerPC SPR Encodings

Register Name	SPR ¹			Access	mfspir/mtspir
	Decimal	spr[5–9]	spr[0–4]		
DABR ²	1013	11111	10101	Supervisor (OEA)	Both
DAR	19	00000	10011	Supervisor (OEA)	Both
DBAT0L	537	10000	11001	Supervisor (OEA)	Both
DBAT0U	536	10000	11000	Supervisor (OEA)	Both
DBAT1L	539	10000	11011	Supervisor (OEA)	Both
DBAT1U	538	10000	11010	Supervisor (OEA)	Both
DBAT2L	541	10000	11101	Supervisor (OEA)	Both
DBAT2U	540	10000	11100	Supervisor (OEA)	Both
DBAT3L	543	10000	11111	Supervisor (OEA)	Both
DBAT3U	542	10000	11110	Supervisor (OEA)	Both
DEC	22	00000	10110	Supervisor (OEA)	Both
DSISR	18	00000	10010	Supervisor (OEA)	Both
EAR ²	282	01000	11010	Supervisor (OEA)	Both
IBAT0L	529	10000	10001	Supervisor (OEA)	Both
IBAT0U	528	10000	10000	Supervisor (OEA)	Both
IBAT1L	531	10000	10011	Supervisor (OEA)	Both
IBAT1U	530	10000	10010	Supervisor (OEA)	Both
IBAT2L	533	10000	10101	Supervisor (OEA)	Both
IBAT2U	532	10000	10100	Supervisor (OEA)	Both
IBAT3L	535	10000	10111	Supervisor (OEA)	Both
IBAT3U	534	10000	10110	Supervisor (OEA)	Both

Table 2-74. Supervisor-Level PowerPC SPR Encodings (continued)

Register Name	SPR ¹			Access	mfspr/mtspr
	Decimal	spr[5–9]	spr[0–4]		
MMCR0 ²	952	11101	11000	Supervisor (OEA)	Both
MMCR1 ²	956	11101	11100	Supervisor (OEA)	Both
PIR ²	1023	11111	11111	Supervisor (OEA)	Both
PMC1 ²	953	11101	11001	Supervisor (OEA)	Both
PMC2 ²	954	11101	11010	Supervisor (OEA)	Both
PMC3 ²	957	11101	11101	Supervisor (OEA)	Both
PMC4 ²	958	11101	11110	Supervisor (OEA)	Both
PMC5 ²	945	11101	10001	Supervisor (OEA)	Both
PMC6 ²	946	11101	10010	Supervisor (OEA)	Both
PVR	287	01000	11111	Supervisor (OEA)	mfspr
SDR1	25	00000	11001	Supervisor (OEA)	Both
SIAR ²	955	11101	11011	Supervisor (OEA)	Both
SPRG0	272	01000	10000	Supervisor (OEA)	Both
SPRG1	273	01000	10001	Supervisor (OEA)	Both
SPRG2	274	01000	10010	Supervisor (OEA)	Both
SPRG3	275	01000	10011	Supervisor (OEA)	Both
SRR0	26	00000	11010	Supervisor (OEA)	Both
SRR1	27	00000	11011	Supervisor (OEA)	Both
TBL ³	284	01000	11100	Supervisor (OEA)	mtspr
TBU ³	285	01000	11101	Supervisor (OEA)	mtspr

¹ Note that the order of the two 5-bit halves of the SPR number is reversed compared with actual instruction coding. For **mtspr** and **mfspr** instructions, the SPR number coded in assembly language does not appear directly as a 10-bit binary number in the instruction. The number coded is split into two 5-bit halves that are reversed in the instruction, with the high-order 5 bits appearing in bits 16–20 of the instruction and the low-order 5 bits in bits 11–15.

² Optional register defined by the PowerPC architecture.

³ The TB registers are referred to as TBRs rather than SPRs and can be written to using the **mtspr** instruction in supervisor mode and the TBR numbers here. The TB registers can be read in user mode using the **mftb** instruction and specifying TBR 268 for TBL and TBR 269 for TBU.

Encodings for the supervisor-level e600-specific SPRs are listed in [Table 2-65](#).

Table 2-75. Supervisor-Level SPR Encodings for e600-Defined Registers

Register Name	SPR ¹			Access	mfspr/mtspr
	Decimal	spr[5–9]	spr[0–4]		
BAMR	951	11101	10111	Supervisor (OEA)	Both
DBAT4L ²	569	10001	11001	Supervisor (OEA)	Both
DBAT4U ²	568	10001	11000	Supervisor (OEA)	Both
DBAT5L ²	571	10001	11011	Supervisor (OEA)	Both
DBAT5U ²	570	10001	11010	Supervisor (OEA)	Both
DBAT6L ²	573	10001	11101	Supervisor (OEA)	Both
DBAT6U ²	572	10001	11100	Supervisor (OEA)	Both
DBAT7L ²	575	10001	11111	Supervisor (OEA)	Both
DBAT7U ²	574	10001	11110	Supervisor (OEA)	Both
HID0	1008	11111	10000	Supervisor (OEA)	Both
HID1	1009	11111	10001	Supervisor (OEA)	Both
IABR	1010	11111	10010	Supervisor (OEA)	Both
IBAT4L ²	561	10001	10001	Supervisor (OEA)	Both
IBAT4U ²	560	10001	10000	Supervisor (OEA)	Both
IBAT5L ²	563	10001	10011	Supervisor (OEA)	Both
IBAT5U ²	562	10001	10010	Supervisor (OEA)	Both
IBAT6L ²	565	10001	10101	Supervisor (OEA)	Both
IBAT6U ²	564	10001	10100	Supervisor (OEA)	Both
IBAT7L ²	567	10001	10111	Supervisor (OEA)	Both
IBAT7U ²	566	10001	10110	Supervisor (OEA)	Both
ICTC	1019	11111	11011	Supervisor (OEA)	Both
ICTRL	1011	11111	10011	Supervisor (OEA)	Both
L2CR	1017	11111	11001	Supervisor (OEA)	Both
L2ERRADDR ²	995	11111	00011	Supervisor (OEA)	mfspr
L2ERRATTR ²	994	11111	00010	Supervisor (OEA)	Both
L2ERREADDR ²	996	11111	00100	Supervisor (OEA)	mfspr
L2CAPTDATAHI ²	988	11110	11100	Supervisor (OEA)	mfspr
L2CAPTDATALO ²	989	11110	11101	Supervisor (OEA)	mfspr
L2CAPTECC ²	990	11110	11110	Supervisor (OEA)	mfspr
L2ERRCTL ²	997	11111	00101	Supervisor (OEA)	Both
L2ERRINJCTL ²	987	11110	11011	Supervisor (OEA)	Both

Table 2-75. Supervisor-Level SPR Encodings for e600-Defined Registers (continued)

Register Name	SPR ¹			Access	mfspr/mtspr
	Decimal	spr[5–9]	spr[0–4]		
L2ERRINJHI ²	985	11110	11001	Supervisor (OEA)	Both
L2ERRINJLO ²	986	11110	11010	Supervisor (OEA)	Both
L2ERRDET ²	991	11110	11111	Supervisor (OEA)	Special ³
L2ERRDIS ²	992	11111	00000	Supervisor (OEA)	Both
L2ERRINTEN ²	993	11111	00001	Supervisor (OEA)	Both
LDSTCR	1016	11111	1000	Supervisor (OEA)	Both
MMCR2	944	11101	10000	Supervisor (OEA)	Both
MSSCR0	1014	11111	10110	Supervisor (OEA)	Both
MSSSR0	1015	11111	10111	Supervisor (OEA)	Both
PTEHI	981	11110	10101	Supervisor (OEA)	Both
PTELO	982	11110	10110	Supervisor (OEA)	Both
SPRG4 ²	276	01000	10100	Supervisor (OEA)	Both
SPRG5 ²	277	01000	10101	Supervisor (OEA)	Both
SPRG6 ²	278	01000	100110	Supervisor (OEA)	Both
SPRG7 ²	279	01000	10111	Supervisor (OEA)	Both
SVR	286	01000	11110	Supervisor (OEA)	mfspr
TLBMISS	980	11110	10100	Supervisor (OEA)	Both

¹ Note that the order of the two 5-bit halves of the SPR number is reversed compared with actual instruction coding. For **mtspr** and **mfspr** instructions, the SPR number coded in assembly language does not appear directly as a 10-bit binary number in the instruction. The number coded is split into two 5-bit halves that are reversed in the instruction, with the high-order 5 bits appearing in bits 16–20 of the instruction and the low-order 5 bits in bits 11–15.

² e600-specific register that may not be supported on other processors that implement the PowerPC architecture.

³ Most bits are bit reset/write 1 clear. A write of 0 to a bit does not change it. A write of 1 to a bit clears it. Reads act normally.

2.3.6.3 Memory Control Instructions—OEA

Memory control instructions include the following:

- Cache management instructions (supervisor-level and user-level)
- Translation lookaside buffer management instructions

This section describes supervisor-level memory control instructions. [Section 2.3.5.3, “Memory Control Instructions—VEA,”](#) describes user-level memory control instructions.

2.3.6.3.1 Supervisor-Level Cache Management Instruction—(OEA)

Table 2-76 lists the only supervisor-level cache management instruction.

Table 2-76. Supervisor-Level Cache Management Instruction

Name	Mnemonic	Syntax	Implementation Notes
Data Cache Block Invalidate	dcbi	rA,rB	The dcbi instruction is executed identically to the dcbf instruction except that it is privileged (supervisor-only). See Section 2.3.5.3.1, “User-Level Cache Instructions—VEA.”

See Section 2.3.5.3.1, “User-Level Cache Instructions—VEA,” for cache instructions that provide user-level programs the ability to manage the caches in the core. If the effective address references a direct-store segment, the instruction is treated as a no-op.

2.3.6.3.2 Translation Lookaside Buffer Management Instructions—OEA

The address translation mechanism is defined in terms of the segment descriptors and page table entries (PTEs) that processors use to locate the logical-to-physical address mapping for a particular access. These segment descriptors and PTEs reside in segment registers within the core and page tables in memory, respectively.

Implementation Note—The e600 core provides two implementation-specific instructions (**tlbld** and **tlbli**) that are used by software table search operations following TLB misses to load TLB entries within the core when $HID0[STEN] = 1$.

For more information on **tlbld** and **tlbli** refer to Section 2.3.8, “Implementation-Specific Instructions.”

See Chapter 5, “Memory Management,” for more information about TLB operations. Table 2-77 summarizes the operation of the TLB instructions in the e600 core. Note that the broadcast of **tlbie** and **tlbsync** instructions is enabled by the setting of $HID1[SYNCBE]$.

Table 2-77. Translation Lookaside Buffer Management Instruction

Name	Mnemonic	Syntax	Implementation Notes
TLB Invalidate Entry	tlbie	rB	Invalidates both ways in both instruction and data TLB entries at the index provided by EA[14–19]. It executes regardless of the MSR[DR] and MSR[IR] settings. To invalidate all entries in both TLBs, the programmer should issue 64 tlbie instructions that each successively increment this field.
Load Data TLB Entry	tlbld	rB	Load Data TLB Entry Loads fields from the PTEHI and PTELO and the EA in rB to the way defined in rB[31].
Load Instruction TLB Entry	tlbli	rB	Load Instruction TLB Entry Loads fields from the PTEHI and PTELO and the EA in rB to the way defined in rB[31].
TLB Synchronize	tlbsync	—	TLBSYNC is broadcast.

Implementation Note—The **tlbia** instruction is optional for an implementation if its effects can be achieved through some other mechanism. Therefore, it is not implemented on the e600 core. As described above, **tlbie** can be used to invalidate a particular index of the TLB based on EA[14–19]—a sequence of

64 **tlbie** instructions followed by a **tlbsync** instruction invalidates all the TLB structures (for EA[14–19] = 0, 1, 2, . . . , 63). Attempting to execute **tlbia** causes an illegal instruction program interrupt.

The presence and exact semantics of the TLB management instructions are implementation-dependent. To minimize compatibility problems, system software should incorporate uses of these instructions into subroutines.

2.3.7 Recommended Simplified Mnemonics

The description of each instruction includes the mnemonic and a formatted list of operands. PowerPC-architecture-compliant assemblers support the mnemonics and operand lists. To simplify assembly language programming, a set of simplified mnemonics and symbols is provided for some of the most frequently-used instructions; refer to Appendix F, “Simplified Mnemonics,” in the *Programming Environments Manual* for a complete list. Programs written to be portable across the various assemblers for the PowerPC architecture should not assume the existence of mnemonics not described in this document.

2.3.8 Implementation-Specific Instructions

This section provides the details for the two e600 core implementation-specific instructions—**tlbld** and **tbli**.

tlbld

Load Data TLB Entry

tlbld

Integer Unit

tlbld

rB

Reserved



$EA \leftarrow (rB)$

TLB entry created from PTEHI and PTELO

DTLB entry selected by $EA[14-19]$ and $rB[31] \leftarrow$ created TLB entry

The EA is the contents of **rB**. The **tlbld** instruction loads the contents of the PTEHI special purpose register and PTELO special purpose register into the selected data TLB entry. The set of the data TLB to be loaded is determined by $EA[14-19]$. The way to be loaded is determined by $rB[31]$. $EA[10-13]$ are stored in the tag portion of the TLB and are used to match a new EA when a new EA is being translated.

The **tlbld** instruction should only be executed when address translation is disabled ($MSR[IR] = 0$ and $MSR[DR] = 0$).

Note that it is possible to execute the **tlbld** instruction when address translation is enabled; however, extreme caution should be used in doing so. If data address translation is enabled ($MSR[DR] = 1$), **tlbld** must be preceded by a **sync** instruction and succeeded by a context synchronizing instruction.

Note that if extended addressing is not enabled ($HID0[XAEN] = 0$), then $PTELO[20-22]$ and $PTELO[29]$ should be cleared (zero) by software when executing a **tlbld** instruction.

This is a supervisor-level instruction; it is also a e600-specific instruction, and not part of the PowerPC instruction set.

Other registers altered:

- None

tlbli

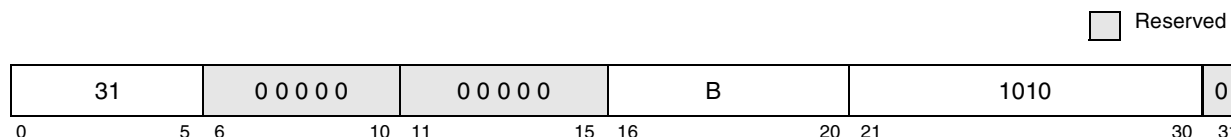
Load Instruction TLB Entry

tlbli

Integer Unit

tlbli

rB



EA ← (rB)

TLB entry created from PTEHI and PTELO

ITLB entry selected by EA[14–19] and rB[31] ← created TLB entry

The EA is the contents of **rB**. The **tlbli** instruction loads an instruction TLB entry. The **tlbli** instruction loads the contents of the PTEHI special purpose register and PTELO special purpose register into a selected instruction TLB entry. The set of the instruction TLB to be loaded is determined by EA[14–19]. The way to be loaded is determined by rB[31]. EA[10–13] are stored in the tag portion of the TLB and are used to match a new EA when a new EA is being translated.

The **tlbli** instruction should only be executed when address translation is disabled (MSR[IR] = 0 and MSR[DR] = 0).

Note that it is possible to execute the **tlbli** instruction when address translation is enabled; however, extreme caution should be used in doing so. If instruction address translation is enabled (MSR[IR] = 1), **tlbli** must be followed by a context synchronizing instruction such as **isync** or **rfi**.

Note that if extended addressing is not enabled (HID0[XAEN]=0) then PTELO[20–22] and PTELO[29] should be cleared (set to zero) by software when executing a **tlbli** instruction.

Note also that care should be taken to avoid modification of the instruction TLB entries that translate current instruction prefetch addresses.

This is a supervisor-level instruction; it is also a e600-specific instruction, and not part of the PowerPC instruction set.

Other registers altered:

- None

2.4 AltiVec Instructions

The following sections provide a general summary of the instructions and addressing modes defined by the AltiVec Instruction Set Architecture (ISA). For specific details on the AltiVec instructions see the *AltiVec Technology Programming Environments Manual* and [Chapter 7, “AltiVec Technology Implementation.”](#) AltiVec instructions belong primarily to the UISA, unless otherwise noted. AltiVec instructions are divided into the following categories:

- Vector integer arithmetic instructions—These include arithmetic, logical, compare, rotate and shift instructions, described in [Section 2.3.4.1, “Integer Instructions.”](#)
- Vector floating-point arithmetic instructions—These floating-point arithmetic instructions and floating-point modes are described in [Section 2.3.4.2, “Floating-Point Instructions.”](#)
- Vector load and store instructions—These load and store instructions for vector registers are described in [Section 2.5.3, “Vector Load and Store Instructions.”](#)
- Vector permutation and formatting instructions—These include pack, unpack, merge, splat, permute, select, and shift instructions, and are described in [Section 2.5.5, “Vector Permutation and Formatting Instructions.”](#)
- Processor control instructions—These instructions are used to read and write from the AltiVec status and control register, and are described in [Section 2.3.4.6, “Processor Control Instructions—UISA.”](#)
- Memory control instructions—These instructions are used for managing caches (user level and supervisor level), and are described in [Section 2.6.1, “AltiVec Vector Memory Control Instructions—VEA.”](#)

This grouping of instructions does not necessarily indicate the execution unit that processes a particular instruction or group of instructions within a processor implementation.

Integer instructions operate on byte, half-word, and word operands. Floating-point instructions operate on single-precision operands. The AltiVec ISA uses instructions that are four bytes long and word-aligned. It provides for byte, half-word, word, and quad-word operand fetches and stores between memory and the vector registers (VRs).

Arithmetic and logical instructions do not read or modify memory. To use the contents of a memory location in a computation and then modify the same or another memory location, the memory contents must be loaded into a register, modified, and then written to the target location using load and store instructions.

The AltiVec ISA supports both big-endian and little-endian byte ordering. The default byte and bit ordering is big-endian; see “Byte Ordering,” in Chapter 3, “Operand Conventions,” of the *AltiVec Technology Programming Environments Manual* for more information.

2.5 AltiVec UISA Instructions

This section describes the instructions defined in the AltiVec user instruction set architecture (UISA).

2.5.1 Vector Integer Instructions

The following are categories for vector integer instructions:

- Vector integer arithmetic instructions
- Vector integer compare instructions
- Vector integer logical instructions
- Vector integer rotate and shift instructions

Integer instructions use the content of VRs as source operands and also place results into VRs. Setting the Rc bit of a vector compare instruction causes the CR6 field of the PowerPC condition register (CR) to be updated; refer to [Section 2.5.1.2, “Vector Integer Compare Instructions,”](#) for more details.

The AltiVec integer instructions treat source operands as signed integers unless the instruction is explicitly identified as performing an unsigned operation. For example, both the Vector Add Unsigned Word Modulo (**vadduwm**) and Vector Multiply Odd Unsigned Byte (**vmuloub**) instructions interpret the operands as unsigned integers.

2.5.1.1 Vector Integer Arithmetic Instructions

[Table 2-78](#) lists the integer arithmetic instructions for the processors that implement the PowerPC architecture.

Table 2-78. Vector Integer Arithmetic Instructions

Name	Mnemonic	Syntax
Vector Add Unsigned Integer [b,h,w] Modulo1	vaddubm vadduhm vadduwm	vD,vA,vB
Vector Add Unsigned Integer [b,h,w] Saturate	vaddubs vadduhs vadduws	vD,vA,vB
Vector Add Signed Integer [b.h.w] Saturate	vaddsbs vaddshs vaddsws	vD,vA,vB
Vector Add and Write Carry-out Unsigned Word	vaddcuw	vD,vA,vB
Vector Subtract Unsigned Integer Modulo	vsububm vsubuhm vsubuwm	vD,vA,vB
Vector Subtract Unsigned Integer Saturate	vsububs vsubuhs vsubuws	vD,vA,vB

Table 2-78. Vector Integer Arithmetic Instructions (continued)

Name	Mnemonic	Syntax
Vector Subtract Signed Integer Saturate	vsubsbs vsubshs vsubsws	vD,vA,vB
Vector Subtract and Write Carry-out Unsigned Word	vsubcuw	vD,vA,vB
Vector Multiply Odd Unsigned Integer [b,h] Modulo	vmuloub vmulouh	vD,vA,vB
Vector Multiply Odd Signed Integer [b,h] Modulo	vmulosb vmulosh	vD,vA,vB
Vector Multiply Even Unsigned Integer [b,h] Modulo	vmuleub vmuleuh	vD,vA,vB
Vector Multiply Even Signed Integer [b,h] Modulo	vmulesb vmulesh	vD,vA,vB
Vector Multiply-High and Add Signed Half-Word Saturate	vmhaddshs	vD,vA,vB, vC
Vector Multiply-High Round and Add Signed Half-Word Saturate	vmhraddshs	vD,vA,vB,vC
Vector Multiply-Low and Add Unsigned Half-Word Modulo	vmladduhm	vD,vA,vB,vC
Vector Multiply-Sum Unsigned Integer [b,h] Modulo	vmsumubm vmsumuhm	vD,vA,vB,vC
Vector Multiply-Sum Signed Half-Word Saturate	vmsumshs	vD,vA,vB,vC
Vector Multiply-Sum Unsigned Half-Word Saturate	vmsumuhs	vD,vA,vB,vC
Vector Multiply-Sum Mixed Byte Modulo	vmsummbm	vD,vA,vB,vC
Vector Multiply-Sum Signed Half-Word Modulo	vmsumshm	vD,vA,vB,vC
Vector Sum Across Signed Word Saturate	vsumsws	vD,vA,vB
Vector Sum Across Partial (1/2) Signed Word Saturate	vsum2sws	vD,vA,vB
Vector Sum Across Partial (1/4) Unsigned Byte Saturate	vsum4ubs	vD,vA,vB
Vector Sum Across Partial (1/4) Signed Integer Saturate	vsum4sbs vsum4shs	vD,vA,vB
Vector Average Unsigned Integer	vavgub vavguh vavguw	vD,vA,vB
Vector Average Signed Integer	vavgsb vavgsh vavgsw	vD,vA,vB
Vector Maximum Unsigned Integer	vmaxub vmaxuh vmaxuw	vD,vA,vB
Vector Maximum Signed Integer	vmaxsb vmaxsh vmaxsw	vD,vA,vB

Table 2-78. Vector Integer Arithmetic Instructions (continued)

Name	Mnemonic	Syntax
Vector Minimum Unsigned Integer	vminub vminuh vminuw	vD,vA,vB
Vector Minimum Signed Integer	vminsb vminsh vminsw	vD,vA,vB

2.5.1.2 Vector Integer Compare Instructions

The vector integer compare instructions algebraically or logically compare the contents of the elements in vector register **vA** with the contents of the elements in **vB**. Each compare result vector is comprised of TRUE (0xFF, 0xFFFF, 0xFFFF_FFFF) or FALSE (0x00, 0x0000, 0x0000_0000) elements of the size specified by the compare source operand element (byte, half word, or word). The result vector can be directed to any VR and can be manipulated with any of the instructions as normal data (for example, combining condition results).

Vector compares provide equal-to and greater-than predicates. Others are synthesized from these by logically combining or inverting result vectors.

The integer compare instructions (shown in [Table 2-80](#)) can optionally set the CR6 field of the PowerPC condition register. If $R_c = 1$ in the vector integer compare instruction, then CR6 is set to reflect the result of the comparison, as follows in [Table 2-79](#).

Table 2-79. CR6 Field Bit Settings for Vector Integer Compare Instructions

CR Bit	CR6 Bit	Vector Compare
24	0	1 Relation is true for all element pairs (that is, vD is set to all ones)
25	1	0
26	2	1 Relation is false for all element pairs (that is, register vD is cleared)
27	3	0

[Table 2-80](#) summarizes the vector integer compare instructions.

Table 2-80. Vector Integer Compare Instructions

Name	Mnemonic	Syntax
Vector Compare Greater than Unsigned Integer	vcmpgtub [.] vcmpgtuh [.] vcmpgtuw [.]	vD,vA,vB
Vector Compare Greater than Signed Integer	vcmpgtsb [.] vcmpgtsh [.] vcmpgtsw [.]	vD,vA,vB
Vector Compare Equal to Unsigned Integer	vcmpequb [.] vcmpequh [.] vcmpequw [.]	vD,vA,vB

2.5.1.3 Vector Integer Logical Instructions

The vector integer logical instructions shown in [Table 2-81](#) perform bit-parallel operations on the operands.

Table 2-81. Vector Integer Logical Instructions

Name	Mnemonic	Syntax
Vector Logical AND	vand	vD,vA,vB
Vector Logical OR	vor	vD,vA,vB
Vector Logical XOR	vxor	vD,vA,vB
Vector Logical AND with Complement	vandc	vD,vA,vB
Vector Logical NOR	vnor	vD,vA,vB

2.5.1.4 Vector Integer Rotate and Shift Instructions

The vector integer rotate instructions are summarized in [Table 2-82](#).

Table 2-82. Vector Integer Rotate Instructions

Name	Mnemonic	Syntax
Vector Rotate Left Integer	vrlb vrlh vrlw	vD,vA,vB

The vector integer shift instructions are summarized in [Table 2-83](#).

Table 2-83. Vector Integer Shift Instructions

Name	Mnemonic	Syntax
Vector Shift Left Integer	vslb vslh vslw	vD,vA,vB
Vector Shift Right Integer	vsrb vsrh vsrw	vD,vA,vB
Vector Shift Right Algebraic Integer	vsrab vsrah vsraw	vD,vA,vB

2.5.2 Vector Floating-Point Instructions

This section describes the vector floating-point instructions that include the following:

- Vector floating-point arithmetic instructions
- Vector floating-point rounding and conversion instructions
- Vector floating-point compare instructions
- Vector floating-point estimate instructions

The AltiVec floating-point data format complies with the ANSI/IEEE-754 standard as defined for single precision. A quantity in this format represents a signed normalized number, a signed denormalized number, a signed zero, a signed infinity, a quiet not a number (QNaN), or a signaling NaN (SNaN). Operations conform to the description in the section “AltiVec Floating-Point Instructions-UISA,” in Chapter 3, “Operand Conventions,” of the *AltiVec Technology Programming Environments Manual*.

The AltiVec ISA does not report IEEE exceptions but rather produces default results as specified by the Java/IEEE/C9X Standard; for further details on exceptions see “Floating-Point Exceptions,” in Chapter 3, “Operand Conventions,” of the *AltiVec Technology Programming Environments Manual*.

2.5.2.1 Vector Floating-Point Arithmetic Instructions

The floating-point arithmetic instructions are summarized in [Table 2-84](#).

Table 2-84. Vector Floating-Point Arithmetic Instructions

Name	Mnemonic	Syntax
Vector Add Floating-Point	vaddfp	vD,vA,vB
Vector Subtract Floating-Point	vsubfp	vD,vA,vB
Vector Maximum Floating-Point	vmaxfp	vD,vA,vB
Vector Minimum Floating-Point	vminfp	vD,vA,vB

2.5.2.2 Vector Floating-Point Multiply-Add Instructions

Vector multiply-add instructions are critically important to performance because a multiply followed by a data dependent addition is the most common idiom in DSP algorithms. In most implementations, floating-point multiply-add instructions perform with the same latency as either a multiply or add alone, thus doubling performance in comparing to the otherwise serial multiply and adds.

AltiVec floating-point multiply-add instructions fuse (a multiply-add fuse implies that the full product participates in the add operation without rounding, only the final result rounds). This not only simplifies the implementation and reduces latency (by eliminating the intermediate rounding) but also increases the accuracy compared to separate multiply and adds.

The floating-point multiply-add instructions are summarized in [Table 2-85](#).

Table 2-85. Vector Floating-Point Multiply-Add Instructions

Name	Mnemonic	Syntax
Vector Multiply-Add Floating-Point	vmaddfp	vD,vA,vC,vB
Vector Negative Multiply-Subtract Floating-Point	vnmsubfp	vD,vA,vC,vB

2.5.2.3 Vector Floating-Point Rounding and Conversion Instructions

All AltiVec floating-point arithmetic instructions use the IEEE Std. 754 default rounding mode round-to-nearest. The AltiVec ISA does not provide the IEEE Std. 754 directed rounding modes.

The AltiVec ISA provides separate instructions for converting floating-point numbers to integral floating-point values for all IEEE Std. 754 rounding modes as follows:

- Round-to-nearest (**vrfin**) (round)
- Round-toward-zero (**vrfiz**) (truncate)
- Round-toward-minus-infinity (**vrfim**) (floor)
- Round-toward-positive-infinity (**vrfip**) (ceiling)

Floating-point conversions to integers (**vctuxs**, **vctxsx**) use round-toward-zero (truncate) rounding. The floating-point rounding instructions are shown in [Table 2-86](#).

Table 2-86. Vector Floating-Point Rounding and Conversion Instructions

Name	Mnemonic	Syntax
Vector Round to Floating-Point Integer Nearest	vrfin	vD,vB
Vector Round to Floating-Point Integer toward Zero	vrfiz	vD,vB
Vector Round to Floating-Point Integer toward Positive Infinity	vrfip	vD,vB
Vector Round to Floating-Point Integer toward Minus Infinity	vrfim	vD,vB
Vector Convert from Unsigned Fixed-Point Word	vcfux	vD,vB,UIMM
Vector Convert from Signed Fixed-Point Word	vcfsx	vD,vB,UIMM
Vector Convert to Unsigned Fixed-Point Word Saturate	vctuxs	vD,vB,UIMM
Vector Convert to Signed Fixed-Point Word Saturate	vctxsx	vD,vB,UIMM

2.5.2.4 Vector Floating-Point Compare Instructions

The floating-point compare instructions are summarized in [Table 2-87](#).

Table 2-87. Vector Floating-Point Compare Instructions

Name	Mnemonic	Syntax
Vector Compare Greater Than Floating-Point [Record]	vcmpgtfp [.]	vD,vA,vB
Vector Compare Equal to Floating-Point [Record]	vcmpeqfp [.]	vD,vA,vB
Vector Compare Greater Than or Equal to Floating-Point [Record]	vcmpgeqfp [.]	vD,vA,vB
Vector Compare Bounds Floating-Point [Record]	vcmpbfp [.]	vD,vA,vB

2.5.2.5 Vector Floating-Point Estimate Instructions

The floating-point estimate instructions are summarized in [Table 2-88](#).

Table 2-88. Vector Floating-Point Estimate Instructions

Name	Mnemonic	Syntax
Vector Reciprocal Estimate Floating-Point	vrefp	vD,vB
Vector Reciprocal Square Root Estimate Floating-Point	vrsqrtfp	vD,vB

Table 2-88. Vector Floating-Point Estimate Instructions (continued)

Name	Mnemonic	Syntax
Vector Log2 Estimate Floating-Point	vlogefp	vD,vB
Vector 2 Raised to the Exponent Estimate Floating-Point	vexptefp	vD,vB

2.5.3 Vector Load and Store Instructions

Only very basic load and store operations are provided in the AltiVec ISA. This keeps the circuitry in the memory path fast so the latency of memory operations is minimized. Instead, a powerful set of field manipulation instructions are provided to manipulate data into the desired alignment and arrangement after the data has been brought into the VRs.

Load vector indexed (**lvx**, **lvxl**) and store vector indexed (**stvx**, **stvxl**) instructions transfer an aligned quad-word vector between memory and VRs. Load vector element indexed (**lvebx**, **lvehx**, **lvewx**) and store vector element indexed instructions (**stvebx**, **stvehx**, **stvewx**) transfer byte, half-word, and word scalar elements between memory and VRs.

2.5.3.1 Vector Load Instructions

For vector load instructions, the byte, half word, word, or quad word addressed by the EA (effective address) is loaded into vD.

The default byte and bit ordering is big-endian as in the PowerPC architecture; see “Byte Ordering,” in Chapter 3, “Operand Conventions,” of the *AltiVec Technology Programming Environments Manual* for information about little-endian byte ordering.

Table 2-89 summarizes the vector load instructions.

Table 2-89. Vector Integer Load Instructions

Name	Mnemonic	Syntax
Load Vector Element Integer Indexed	lvebx lvehx lvewx	vD,rA,rB
Load Vector Element Indexed	lvx	vD,rA,rB
Load Vector Element Indexed LRU ¹	lvxl	vD,rA,rB

¹ On the e600 core, **lvxl** and **stvxl** are interpreted to be transient. See Section 7.1.2.3, “Data Stream Touch Instructions.”

2.5.3.2 Vector Load Instructions Supporting Alignment

The **lvsl** and **lvsr** instructions can be used to create the permute control vector to be used by a subsequent **vperm** instruction. Let X and Y be the contents of vA and vB specified by **vperm**. The control vector created by **lvsl** causes the **vperm** to select the high-order 16 bytes of the result of shifting the 32-byte value X || Y left by sh bytes (sh = the value in EA[60–63]). The control vector created by **lvsr** causes the **vperm** to select the low-order 16 bytes of the result of shifting X || Y right by sh bytes.

Table 2-90 summarizes the vector alignment instructions.

Table 2-90. Vector Load Instructions Supporting Alignment

Name	Mnemonic	Syntax
Load Vector for Shift Left	lvsl	vD,rA,rB
Load Vector for Shift Right	lvsr	vD,rA,rB

2.5.3.3 Vector Store Instructions

For vector store instructions, the contents of the VR used as a source (**vS**) are stored into the byte, half word, word or quad word in memory addressed by the effective address (**EA**). Table 2-91 provides a summary of the vector store instructions.

Table 2-91. Vector Integer Store Instructions

Name	Mnemonic	Syntax
Store Vector Element Integer Indexed	svetbx svethx svetwx	vS,rA,rB
Store Vector Element Indexed	stvx	vS,rA,rB
Store Vector Element Indexed LRU ¹	stvxl	vS,rA,rB

¹ On the e600 core, **lvxl** and **stvxl** are interpreted to be transient. See Section 7.1.2.3, “Data Stream Touch Instructions.”

2.5.4 Control Flow

AltiVec instructions can be freely intermixed with existing PowerPC instructions to form a complete program. AltiVec instructions provide a vector compare and select mechanism to implement conditional execution as the preferred mechanism to control data flow in AltiVec programs. In addition, AltiVec vector compare instructions can update the condition register thus providing the communication from AltiVec execution units to PowerPC branch instructions necessary to modify program flow based on vector data.

2.5.5 Vector Permutation and Formatting Instructions

Vector pack, unpack, merge, splat, permute, select, and shift instructions can be used to accelerate various vector math operations and vector formatting. Details of these instructions follow.

2.5.5.1 Vector Pack Instructions

Half-word vector pack instructions (**vpkuhum**, **vpkuhus**, **vpkshus**, **vpkshss**) truncate the sixteen half words from two concatenated source operands producing a single result of sixteen bytes (quad word) using either modulo (2^8), 8-bit signed-saturation, or 8-bit unsigned-saturation to perform the truncation. Similarly, word vector pack instructions (**vpkuwum**, **vpkuwus**, **vpkswus**, **vpksws**) truncate the eight words from two concatenated source operands producing a single result of eight half words using modulo (2^{16}), 16-bit signed-saturation, or 16-bit unsigned-saturation to perform the truncation.

Table 2-92 describes the vector pack instructions.

Table 2-92. Vector Pack Instructions

Name	Mnemonic	Syntax
Vector Pack Unsigned Integer [h,w] Unsigned Modulo	vpkuhum vpkuwum	vD, vA, vB
Vector Pack Unsigned Integer [h,w] Unsigned Saturate	vpkuhus vpkuwus	vD, vA, vB
Vector Pack Signed Integer [h,w] Unsigned Saturate	vpkshus vpkswus	vD, vA, vB
Vector Pack Signed Integer [h,w] signed Saturate	vpkshss vpkswss	vD, vA, vB
Vector Pack Pixel	vpkpx	vD, vA, vB

2.5.5.2 Vector Unpack Instructions

Byte vector unpack instructions unpack the 8 low bytes (or 8 high bytes) of one source operand into 8 half words using sign extension to fill the most-significant bytes (MSBs). Half word vector unpack instructions unpack the 4 low half words (or 4 high half words) of one source operand into 4 words using sign extension to fill the MSBs.

Two special purpose forms of vector unpack are provided—the Vector Unpack Low Pixel (**vupklpx**) and the Vector Unpack High Pixel (**vupkhpix**) instructions for 1/5/5/5 α RGB pixels. The 1/5/5/5 pixel vector unpack, unpacks the four low 1/5/5/5 pixels (or four 1/5/5/5 high pixels) into four 32-bit (8/8/8/8) pixels. The 1-bit α element in each pixel is sign extended to 8 bits, and the 5-bit R, G, and B elements are each zero extended to 8 bits.

Table 2-93 describes the unpack instructions.

Table 2-93. Vector Unpack Instructions

Name	Mnemonic	Syntax
Vector Unpack High Signed Integer	vupkhsb vupkhsh	vD, vB
Vector Unpack High Pixel	vupkhpix	vD, vB
Vector Unpack Low Signed Integer	vupklisb vupklisb	vD, vB
Vector Unpack Low Pixel	vupklpx	vD, vB

2.5.5.3 Vector Merge Instructions

Byte vector merge instructions interleave the 8 low bytes or 8 high bytes from two source operands producing a result of 16 bytes. Similarly, half-word vector merge instructions interleave the 4 low half words (or 4 high half words) of two source operands producing a result of 8 half words, and word vector merge instructions interleave the 2 low words or 2 high words from two source operands producing a result

of 4 words. The vector merge instruction has many uses. For example, it can be used to efficiently transpose SIMD vectors. [Table 2-94](#) describes the merge instructions.

Table 2-94. Vector Merge Instructions

Name	Mnemonic	Syntax
Vector Merge High Integer	vmrghb vmrghh vmrghw	vD, vA, vB
Vector Merge Low Integer	vmrglb vmrglh vmrglw	vD, vA, vB

2.5.5.4 Vector Splat Instructions

When a program needs to perform arithmetic vector operations, the vector splat instructions can be used in preparation for performing arithmetic for which one source vector is to consist of elements that all have the same value. Vector splat instructions can be used to move data where it is required. For example to multiply all elements of a vector register (VR) by a constant, the vector splat instructions can be used to splat the scalar into the VR. Likewise, when storing a scalar into an arbitrary memory location, it must be splatted into a VR, and that VR must be specified as the source of the store. This guarantees that the data appears in all possible positions of that scalar size for the store.

Table 2-95. Vector Splat Instructions

Name	Mnemonic	Syntax
Vector Splat Integer	vspltb vsplth vspltw	vD, vB, UIMM
Vector Splat Immediate Signed Integer	vspltisb vspltish vspltisw	vD, SIMM

2.5.5.5 Vector Permute Instructions

Permute instructions allow any byte in any two source VRs to be directed to any byte in the destination vector. The fields in a third source operand specify from which field in the source operands the corresponding destination field is taken. The Vector Permute (**vperm**) instruction is a very powerful one that provides many useful functions. For example, it provides a way to perform table-lookups and data alignment operations. An example of how to use the **vperm** instruction in aligning data is described in “Quad-Word Data Alignment” in Chapter 3, “Operand Conventions,” of the *AltiVec Technology Programming Environments Manual*. [Table 2-92](#) describes the vector permute instruction.

Table 2-96. Vector Permute Instruction

Name	Mnemonic	Syntax
Vector Permute	vperm	vD, vA,vB,vC

2.5.5.6 Vector Select Instruction

Data flow in the vector unit can be controlled without branching by using a vector compare and the Vector Select (**vsel**) instructions. In this use, the compare result vector is used directly as a mask operand to vector select instructions. The **vsel** instruction selects one field from one or the other of two source operands under control of its mask operand. Use of the TRUE/FALSE compare result vector with select in this manner produces a two instruction equivalent of conditional execution on a per-field basis. [Table 2-97](#) describes the **vsel** instruction.

Table 2-97. Vector Select Instruction

Name	Mnemonic	Syntax
Vector Select	vsel	vD,vA,vB,vC

2.5.5.7 Vector Shift Instructions

The vector shift instructions shift the contents of one or of two VRs left or right by a specified number of bytes (**vslo**, **vsro**, **vsldoi**) or bits (**vsl**, **vsr**). Depending on the instruction, this shift count is specified either by low-order bits of a VR or by an immediate field in the instruction. In the former case the low-order 7 bits of the shift count register give the shift count in bits ($0 \leq \text{count} \leq 127$). Of these 7 bits, the high-order 4 bits give the number of complete bytes by which to shift and are used by **vslo** and **vsro**; the low-order 3 bits give the number of remaining bits by which to shift and are used by **vsl** and **vsr**.

[Table 2-98](#) describes the vector shift instructions.

Table 2-98. Vector Shift Instructions

Name	Mnemonic	Syntax
Vector Shift Left	vsl	vD,vA,vB
Vector Shift Right	vsr	vD,vA,vB
Vector Shift Left Double by Octet Immediate	vsldoi	vD,vA,vB,SH
Vector Shift Left by Octet	vslo	vD,vA,vB
Vector Shift Right by Octet	vsro	vD,vA,vB

2.5.5.8 Vector Status and Control Register Instructions

[Table 2-99](#) summarizes the instructions for reading from or writing to the AltiVec status and control register (VSCR), described in [Section 7.1.1.5, “Vector Save/Restore Register \(VRSAVE\).”](#)

Table 2-99. Move To/From VSCR Register Instructions

Name	Mnemonic	Syntax
Move to AltiVec Status and Control Register	mtvscr	vB
Move from AltiVec Status and Control Register	mfvscr	vB

2.6 Altivec VEA Instructions

The PowerPC virtual environment architecture (VEA) describes the semantics of the memory model that can be assumed by software processes, and includes descriptions of the cache model, cache-control instructions, address aliasing, and other related issues. Implementations that conform to the VEA also adhere to the UISA, but may not necessarily adhere to the OEA. For further details, see Chapter 4, “Addressing Mode and Instruction Set Summary,” in the *Programming Environments Manual*.

This section describes the additional instructions that are provided by the Altivec ISA for the VEA.

2.6.1 Altivec Vector Memory Control Instructions—VEA

Memory control instructions include the following types:

- Cache management instructions (user-level and supervisor-level)
- Translation lookaside buffer (TLB) management instructions

This section briefly summarizes the user-level cache management instructions defined by the Altivec VEA. See [Chapter 3, “L1 and L2 Cache Operation,”](#) for more information about supervisor-level cache, segment register manipulation, and TLB management instructions.

The Altivec architecture specifies the data stream touch instructions **dst(t)**, **dstst(t)**, and it specifies two data stream stop (**dss(all)**) instructions. The e600 core implements all of them. The term **dstx** used below refers to all of the stream touch instructions.

The instructions summarized in this section provide user-level programs the ability to manage caches within the core, see [Chapter 3, “L1 and L2 Cache Operation,”](#) for more information about cache topics.

Bandwidth between the processor and memory is managed explicitly by the programmer through the use of cache management instructions. These instructions provide a way for software to communicate to the cache hardware how it should prefetch and prioritize the writeback of data. The principal instruction for this purpose is a software directed cache prefetch instruction called data stream touch (**dst**). Other related instructions are provided for complete control of the software directed cache prefetch mechanism.

[Table 2-100](#) summarizes the directed prefetch cache instructions defined by the Altivec VEA. Note that these instructions are accessible to user-level programs.

Table 2-100. Altivec User-Level Cache Instructions

Name	Mnemonic	Syntax	Implementation Notes
Data Stream Touch (non-transient)	dst	rA,rB,STRM	—
Data Stream Touch Transient	dstt	rA,rB,STRM	Used for last access
Data Stream Touch for Store	dstst	rA,rB,STRM	Not recommended for use in the e600 core
Data Stream Touch for Store Transient	dststt	rA,rB,STRM	Not recommended for use in the e600 core
Data Stream Stop (one stream)	dss	STRM	—
Data Stream Stop All	dssall	STRM	—

For detailed information for how to use these instruction, see [Section 7.1.2.3, “Data Stream Touch Instructions.”](#)

2.6.2 Altivec Instructions with Specific Implementations for the e600 Core

The Altivec architecture specifies Load Vector Indexed LRU (**lvxl**) and Store Vector Indexed LRU (**stvxl**) instructions. The architecture suggests that these instructions differ from regular Altivec load and store instructions in that they leave cache entries in a least recently used (LRU) state instead of a most recently used (MRU) state. This supports efficient processing of data which is known to have little reuse and poor caching characteristics. The e600 core implements these instructions as suggested. They follow all the cache allocation and replacement policies described in [Section 3.5, “L1 Cache Operation,”](#) but they leave their addressed cache entries in the LRU state. In addition, all LRU instructions are also interpreted to be transient and are also treated as described in [Section 7.1.2.2, “Transient Instructions and Caches](#)

Chapter 3

L1 and L2 Cache Operation

The e600 core contains separate 32-Kbyte, eight-way set-associative level 1 (L1) instruction and data caches to allow the execution units and registers rapid access to instructions and data. In addition, the e600 core features an integrated 1-Mbyte level 2 (L2) cache.

This chapter describes the organization of the L1 instruction and data caches within the core, cache coherency protocols, cache control instructions, various cache operations, and the organization and features of the L2 cache. It describes the interaction between the caches, the load/store unit (LSU), the instruction unit, and the core memory subsystem. This chapter also describes the replacement algorithms used for each of the caches of the e600 core.

Note that in this chapter, the term ‘multiprocessor’ is used in the context of maintaining cache coherency. These multiprocessor devices could be actual processors or other devices that can access system memory, maintain their own caches, and function as bus masters requiring cache coherency.

AltiVec Technology and the Cache Implementation

The implementation of AltiVec technology in the e600 core has implications that affect the cache model. They are as follows:

- AltiVec transient instructions (**dstt**, **dststt**, **lvxl**, and **stvxl**), described in [Section 7.1.2.2, “Transient Instructions and Caches”](#)
- AltiVec LRU instructions (**lvxl**, **stvxl**), described in [Section 3.5.7.3, “AltiVec LRU Instruction Support”](#)
- MPX bus transactions caused by caching-inhibited AltiVec loads and stores or write-through AltiVec stores

The e600 core supports the out-of-order issue of AltiVec instructions. A maximum of two AltiVec instructions can be issued out-of-order to any combination of AltiVec execution units per clock cycle from the bottom two VIQ entries (VIQ1–VIQ0). An instruction in VIQ1 does not have to wait for an instruction in VIQ0 that is waiting for operand availability.

3.1 Overview

The e600 core L1 cache implementation has the following characteristics:

- Separate 32-Kbyte instruction and data caches (Harvard architecture)
- Eight-way set-associative caches
- Caches have 32-byte cache blocks. A cache block is the block of memory that a coherency state describes—it corresponds to a cache line for the L1 data cache.

- Cache directories are physically addressed. The physical (real) address tag is stored in the cache directory.
- Caches implement a pseudo least-recently-used (PLRU) replacement algorithm within each way
- Cache write-back or write-through operation is programmable on a per-page or per-block basis.
- The instruction cache can provide four instructions per clock cycle; the data cache can provide four words per clock cycle.
 - Two-cycle latency and single-cycle throughput for instruction or data cache accesses
- Caches can be disabled in software
- Caches can be locked in software
- Supports a four-state modified/exclusive/shared/invalid (MESI) coherency protocol
 - A single coherency status bit for each instruction cache block allows encoding for the following two possible states:
 - Invalid (INV)
 - Valid (VAL)
 - Three status bits for each data cache block allow encoding for coherency, as follows:
 - 0xx = invalid (I)
 - 101 = shared (S)
 - 100 = exclusive (E)
 - 110 = modified (M)
- Separate copy of data cache tags for efficient snooping
- Both the L1 caches support parity generation and checking (enabled through bits in the ICTRL register) as follows:
 - Instruction cache—one parity bit per instruction
 - Data cache—one parity bit per byte of data
- No snooping of instruction cache except for **icbi** instruction
- The caches implement a pseudo least-recently-used (PLRU) replacement algorithm within each way.
- Data cache supports AltiVec LRU and transient instructions, as described in [Section 1.3.2.2, “AltiVec Instruction Set”](#)
- Critical double- and/or quad-word forwarding is performed as needed. Critical quad-word forwarding is used for AltiVec loads and instruction fetches. Other accesses use critical double-word forwarding.
- Each cache can be invalidated or locked by setting the appropriate bits in the hardware implementation-dependent register 0 (HID0), a special-purpose register (SPR) that is implementation-specific.

The e600 core supports a fully-coherent 64-Gbyte physical memory address space (when extended addressing is enabled with `HID0[XAEN] = 1`). MPX bus snooping ensures the coherency of global memory with respect to the data cache.

On an L1 data cache miss, cache blocks are filled in one 32-byte beat from the L2 cache or MPX bus, and the critical data is forwarded immediately to the requesting execution unit (and register file). Load misses are processed as described in [Section 3.1.2.4, “LSU Load Miss, Castout, and Push Queues,”](#) providing for hits under misses.

Instruction cache blocks are also filled in one 32-byte beat from the L2 cache or MPX bus, and the critical quad word is simultaneously forwarded to the instruction queue, thus minimizing stalls due to cache fill latency. The instruction cache is also not blocked to internal accesses while a cancelled instruction cache miss is outstanding, providing for hits under misses.

The instruction cache provides a 128-bit interface to the instruction unit, so up to four instructions can be made available to the instruction unit in a single clock cycle on an L1 instruction cache hit. The instruction unit accesses the instruction cache frequently in order to sustain the high throughput provided by the 12-entry instruction queue.

Additionally, the L2 cache in the e600 core has the following features:

- Integrated 1-Mbyte, eight-way set-associative unified instruction and data cache
- Maintains instructions, data, or both instructions and data (selectable through L2CR)
- Fully pipelined to provide 32 bytes per two clock cycles to the L1 caches
- Total latency of 11 processor cycles for an L1 data cache miss that hits in the L2 with ECC disabled, 12 cycles with ECC enabled
- Uses one of two random replacement algorithms (selectable through L2CR)
- Cache write-back or write-through operation programmable on a per-page or per-block basis
- Organized as 32 bytes/block and two blocks (sectors)/line (a cache block is the block of memory that a coherency state describes).
- Error correction and detection using a SECDED (single-error correction, double-error detection) protocol. Every 64 bits of data comes with 8 bits of error detection/correction, which can be programmed as either ECC (error correction code) across the 64 bits of data, byte parity, or for no error detection/correction.
- Supports parity generation and checking for both tags and data. Data parity can be enabled through L2CR when ECC is disabled, and tag parity is enabled separately in the L2ERRDIS register.
- Two status bits (MESI[0–1]) for each L2 cache block allow encoding for coherency, as follows:
 - 00 = invalid (I)
 - 01 = shared (S)
 - 10 = exclusive (E)
 - 11 = modified (M)
- Prefetching of the second (unrequired) block through up to three L2 prefetch engines enabled through MSSCR0
- Provides error injection modes for testing

3.1.1 Block Diagram

The L1 instruction and data caches and the L2 cache are integrated in the e600 core as shown in [Figure 3-1](#).

Both L1 caches are tightly coupled with the L2 cache and the core memory subsystem (MSS) to allow efficient access to the L2 cache or the core interface and other bus masters. The core memory subsystem receives requests for memory operations from the LSU (on behalf of the instruction and data caches) and provides queues for loading and storing from the caches.

The core interface performs bus operations on the MPX bus. Depending on the transaction type, the critical 8 bytes (for double words) or 16 bytes (for quad words) are forwarded to the requesting unit. Note that for instruction fetches, the critical quad word is always forwarded. Also, the core interface accumulates 64-bit data beats into a 32-byte entity before loading it into the L1 and L2 caches. The core interface also captures snoop addresses for the L1 data cache, the L2 cache, and the memory reservation (**lwarx** and **stwcx**.) operations.

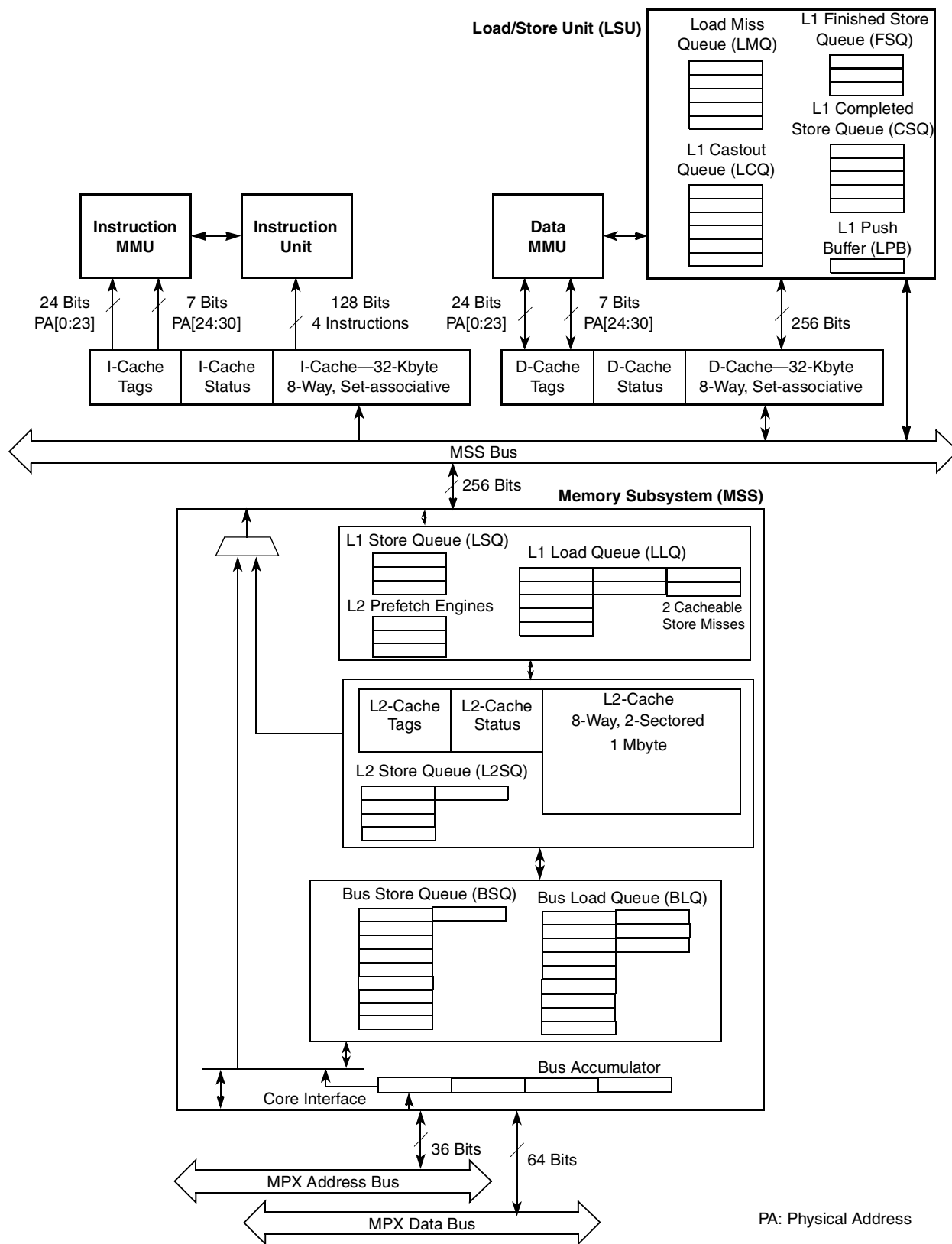


Figure 3-1. Cache/Core Memory Subsystem Integration

3.1.2 Load/Store Unit (LSU)

The data cache supplies data to the general-purpose registers (GPRs), floating-point registers (FPRs), and vector registers (VRs) by means of the load/store unit (LSU). The LSU is directly coupled to the data cache with a 32-byte interface (a cache line) to allow efficient movement of data to and from the GPRs, FPRs, and VRs. The LSU provides all the logic required to calculate effective addresses, handles data alignment to and from the data cache, and provides sequencing for load/store string and load/store multiple operations. Write operations to the data cache can be performed on a byte, half-word, word, double-word, or quad-word basis.

This section describes the LSU queues that support the L1 caches. See [Section 3.3.3, “Load/Store Operations and Architecture Implications,”](#) for more information on architectural coherency implications of load/store operations and the LSU. Also, see [Chapter 6, “Instruction Timing,”](#) for more information on other aspects of the LSU and instruction scheduling considerations.

The vector touch engine (VTE) generates cache line fetch requests based on the contents of the **dst**, **dsts**, **dss**, and **dssall** instructions of the AltiVec specification. These instructions are not disabled by the AltiVec enable bit in the MSR. See [Chapter 7, “AltiVec Technology Implementation,”](#) for more information on the VTE.

3.1.2.1 Cacheable Loads and LSU

When free of data dependencies, cacheable loads execute in the LSU in a speculative manner with a maximum throughput of one per cycle and a 3-cycle latency for integer and vector loads. Note that floating-point loads have a 4-cycle latency through the LSU. Data returned from the cache is held in a rename buffer until the completion logic commits the value to the processor state.

3.1.2.2 LSU Store Queues

Stores cannot be executed speculatively. Stores must be held in the 3-entry finished store queue (FSQ), as shown in [Figure 3-1](#), until the completion logic signals that the store instruction is to be committed. When the store is committed, it moves to the 5-entry completed store queue (CSQ). A store remains in the CSQ until the data cache is updated, if the access is cacheable. If a store is cache-inhibited, the operation moves through the CSQ on to the rest of the core memory subsystem.

To reduce the latency of loads dependent on stores, the e600 core supports data forwarding from any entry in the CSQ before the data is actually written to the cache. The addresses of subsequent loads are compared to all entries in the CSQ and, on a hit, use the data from the newest matching entry. If a load aliases to both a CSQ entry and an FSQ entry, the LSU pipeline stalls. The load needs the newest data from the FSQ and the data is not available until it is completed and moves to the CSQ. Note that no forwarding occurs from a **stwcx.** operation but forwarding does occur from store operations caused by **dcbz** instructions.

3.1.2.3 Store Gathering/Merging

To increase external bandwidth to frame buffers and I/O devices, the e600 core performs store gathering of unguarded write-through stores or unguarded cache-inhibited stores. Two of these store operations are gathered in the CSQ if the following requirements are met:

- Entry CSQ0 is currently accessing the core memory subsystem (that is, it missed in the data cache).
- The stores are bytes, half words, words, double words or quad words (and are the same size).
- The stores are adjacent or overlapping in address (words in the same double word, double words in the same quad word, or quad words in the same cache line).
- The stores are adjacent in the CSQ.
- Both stores are aligned.

The same store-gathering mechanism is used to gather cacheable write-back stores. In this case, these stores can be gathered anywhere within the same cache line if they have not yet accessed the cache. Also, these stores do not need to be of the same size.

Not all stores are gathered. In particular, when there is a series of stores, the first store often appears to the core memory subsystem as ungathered.

Store gathering and store merging are enabled through HID0[SGE]. Note that in addition to the clearing of SGE, the **eieio** instruction may also be used to keep stores from being gathered. If an **eieio** instruction is detected in the store queues, store gathering is not performed. If HID1[SYNCBE] = 1, the **eieio** instruction also causes a MPX bus broadcast operation, which may be used to prevent blocks external to the core, such as bus bridge chips, from gathering stores. See [Section 3.3.3.3, “Load Ordering with Respect to Other Loads,”](#) for more information on the effects of **eieio**.

If multiple cacheable stores are gathered such that the result is one 32-byte store, the processor issues a single line kill block transfer instead of the store to the MPX bus.

3.1.2.4 LSU Load Miss, Castout, and Push Queues

The LSU requests cache blocks that miss in the L1 data cache from the next levels of memory. In the case of a cache miss for a load, the load is placed in the 5-entry load miss queue (LMQ) until it can be serviced to allow for subsequent loads to continue to propagate through the LSU.

The LSU also maintains a 6-entry L1 castout queue (LCQ) as a place-holder for data cache castouts caused by the PLRU replacement algorithm until they can be serviced. Note that castouts are only selected (by the replacement algorithm) when the new cache line is ready to be loaded into the L1. Because all L1 data cache misses can potentially require a castout, misses do not access the L2 or MPX bus until a slot is available in the LCQ for the potential castout operation.

Finally, the LSU also maintains a 1-entry push buffer (LPB) for holding a cache push operation caused by a snoop hit of modified data in the L1 data cache until it can complete. Note that all entries in the LCQ and LPB are snooped when other masters are accessing the MPX bus.

3.1.3 Core Memory Subsystem Blocks

As shown in [Figure 3-1](#), the core memory subsystem interfaces to the L1 instruction and data caches and the LSU with a 256-bit memory subsystem (MSS) bus. The three major logic blocks are described in the following subsections. Conceptually, the general flow for transactions through the core memory subsystem can be considered to be from the L1 service queues, to the L2 cache, to the MPX bus service queues, noting that data from the MPX bus can flow directly from the bus accumulator at the core interface to the 256-bit MSS bus (loading the L2 cache in parallel). Exceptions to this are noted in the following subsections.

Note that transactions on the MPX bus performed by alternate masters are snooped by all relevant entities in the e600 core. Thus the L1 data cache, LSU queues, core memory subsystem queues, and L2 cache are all checked for a snoop hit. When a snoop hit occurs and a push is required, the e600 core retries the bus transaction and performs the push operation or performs data intervention (if `MSSCR0[EIDIS] = 0`).

3.1.3.1 L1 Service Queues

Separate from the LMQ and the two store queues of the LSU, the core memory subsystem block maintains two additional queues for handling L1 misses. The L1 load queue (LLQ) of the core memory subsystem contains a total of nine entries. They are as follows:

- Five for load misses, including those generated by **dcbt**, **dcbtst**, **dst**, and **dsts** instructions
- Two for instruction fetches
- Two for two cacheable store requests that are marked as write-back (`W = 0`), which require a read-with-intent-to-modify load transaction on the MPX bus, or for loads generated by **dcba** and **dcbz**.

For efficiency, these accesses are sent to the L2 cache from the LLQ and reside in the LLQ until the data has been loaded. If the access requires a core interface transaction (based on the L2 response), the LLQ causes that bus transaction to occur. If the access is non-transient and misses in both the L1 and L2 caches, both caches (if enabled) are loaded with the missed data when it is read from the MPX bus.

Also, separate from the L1 store queues of the LSU, the core memory subsystem has an L1 store queue (LSQ) that maintains three entries waiting to be written to the L2 cache (if enabled). The three entries are dedicated as follows:

- One for
 - Stores, including caching-inhibited and write-through stores,
 - Memory management instructions, and
 - **sync**, cache control, and memory synchronization instructions
- One for castouts
- One for snoop pushes

All accesses that go through the LSQ also propagate to the MPX bus, except for L1 castouts that are caused by a replacement operation due to a reload and result in a hit in the L2 cache.

Thus, note that castouts caused by the **dcbf** instruction do propagate to the MPX bus.

3.1.3.2 L2 Cache Block

The integrated L2 cache on the e600 core is a unified (possibly containing instructions and data), 1-Mbyte cache. It is eight-way set-associative and organized with 32-byte blocks and two blocks per line as shown in [Figure 3-17](#). Thus each line shares the same address tag, but the MESI bits are independently maintained for each block.

When the L2 and L1 data cache are enabled, load and store entries from the LLQ and LSQ propagate to the L2 cache, provided caching is allowed (the I bit of WIMG for that particular access is cleared). The L2 services accesses from the LLQ and LSQ with a 5-cycle total latency with ECC disabled (6 cycles with ECC enabled) and a maximum throughput of one L2 access per two clock cycles.

As described in [Section 3.1.3.1, “L1 Service Queues,”](#) LLQ accesses are sent to the L2 cache. LSQ accesses serviced by the L2 that need service by the MPX bus propagate to the L2 store queue (L2SQ) for service by the MPX bus. The L2SQ has a total of 5 entries as follows:

- Four entries for L2 castouts (or stores)
- One entry for pushes and interventions caused by snoop hits

For more detailed information about the functions of the L2 cache, see [Section 3.6, “L2 Cache.”](#)

3.1.3.3 Core Interface Block

As described in [Section 3.1.3.1, “L1 Service Queues,”](#) the LLQ can cause MPX bus transactions to occur. In addition, the core interface block of the core memory subsystem maintains the following two entities that can cause MPX bus transactions:

- Bus store queue (BSQ)—After the L2 cache responds to an access and the access generates a castout (or write-through store) or a push operation, it is sent to the BSQ for service by the core interface. The BSQ maintains up to five outstanding castout operations and/or six push operations.
- L2 prefetch engines—When only one block of an L2 cache line is valid (due to an L2 reload caused by a read miss in the L1 and L2 caches), the L2 prefetch engines can initiate an MPX bus transaction to fill the second block of that L2 cache line. Up to three separate outstanding L2 prefetches can be enabled. See [Section 3.6.3.2, “L2 Prefetch Engines and MSSCR0,”](#) for more detailed information about the L2 prefetch engines.

Also, the core interface block maintains a bus accumulator that collects four double words (instructions or data) from the core interface for forwarding to the MSS bus on reads.

3.2 L1 Cache Organizations

The L1 instruction and data caches of the e600 core are both organized as 128 sets of eight blocks with 32 bytes in each cache line. The following subsections describe the differences in the organization of the instruction and data caches. For information on L2 cache operation, see [Section 3.6, “L2 Cache.”](#)

3.2.1 L1 Data Cache Organization

The L1 data cache is organized as shown in [Figure 3-2](#).

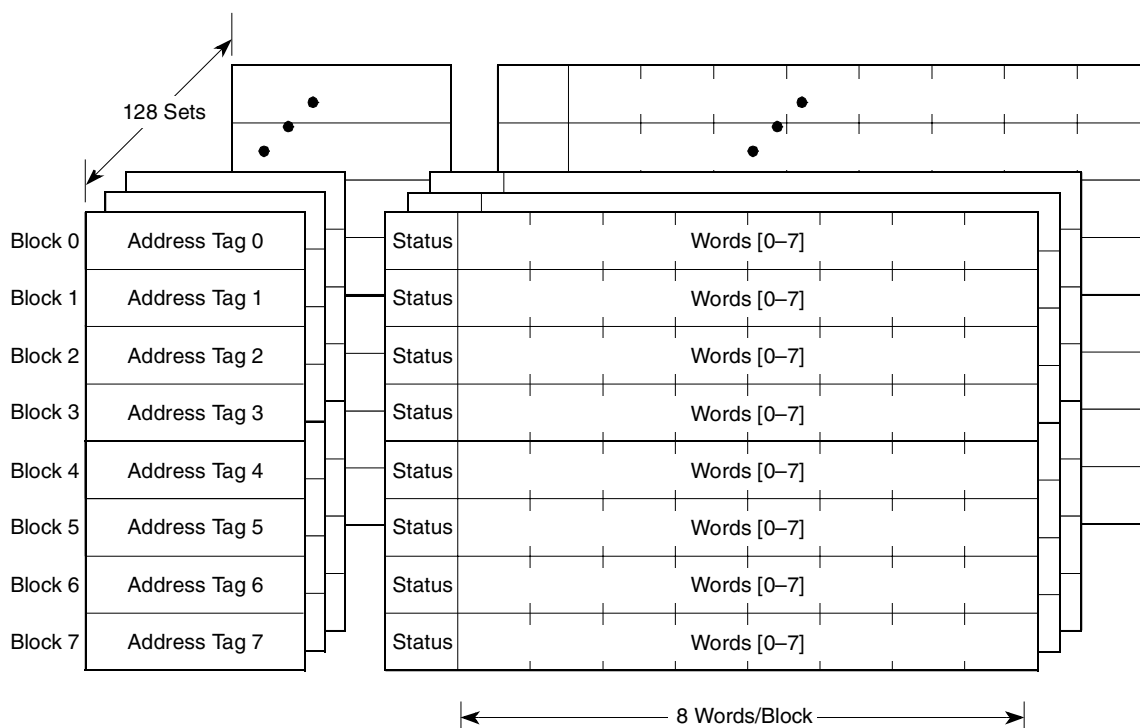


Figure 3-2. L1 Data Cache Organization

Each block consists of 32 bytes of data, three status bits, and an address tag. Note that in the PowerPC architecture, the term ‘cache block,’ or simply ‘block,’ when used in the context of cache implementations, refers to the unit of memory at which coherency is maintained. For the e600 L1 data cache, this is the 32-byte cache line. This value may be different for other implementations using the PowerPC architecture. Also, although it is not shown in [Figure 3-2](#), the data cache has one parity bit/byte (four parity bits/word).

Each cache block contains eight contiguous words from memory that are loaded from an eight-word boundary (that is, bits PA[31:35] of the physical addresses are zero); as a result, cache blocks are aligned with page boundaries. Address bits PA[24:30] provide the index to select a cache set. The tags consist of physical address bits PA[0:23]. Address translation occurs in parallel with set selection (from PA[24:30]). Lower address bits PA[31:35] locate a byte within the selected block. All of these address ranges are shown for 36-bit physical addressing (enabled when HID0[XAEN] = 1). When 32-bit addressing is used (HID0[XAEN] = 0), all of these physical address bits are shifted down by 4, and the tags consist of physical address bits PA[0:19].

The data cache tags are dual-ported and non-blocking for efficient load/store and snooping operations. Thus the data cache can be accessed internally while a load for a miss is pending (allowing hits under misses). When the load miss is actually updating the cache, subsequent loads are blocked for two cycles and stores are blocked for one cycle (but the data for the load miss can be forwarded to the execution unit simultaneously). The LMQ allows misses under misses to occur.

Three status bits are associated with each cache block. These bits are used to implement the modified/exclusive/shared/invalid (MESI) cache coherency protocol. The coherency protocols are described in [Section 3.3, “Memory and Cache Coherency.”](#)

3.2.2 L1 Instruction Cache Organization

The L1 instruction cache is organized as shown in [Figure 3-3](#).

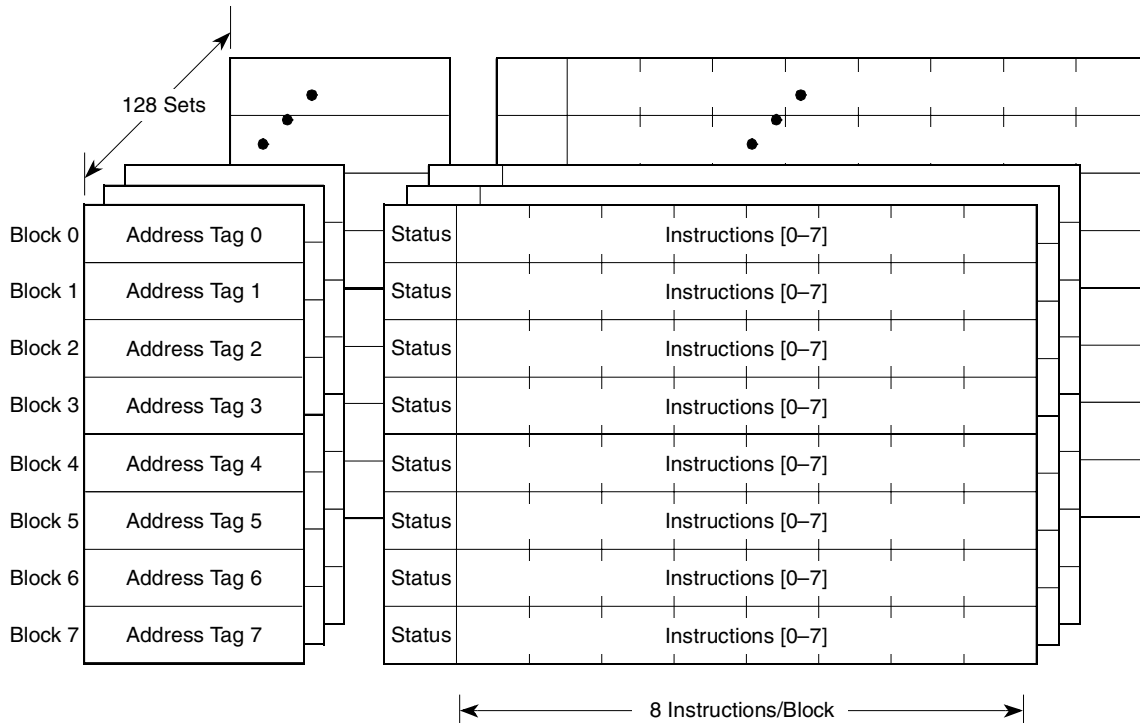


Figure 3-3. L1 Instruction Cache Organization

Each block consists of eight instructions, a single status bit, and an address tag. As with the data cache, each instruction cache block is loaded from an eight-word boundary (that is, bits PA[31:35] of the physical addresses are zero); as a result, cache blocks are aligned with page boundaries. Also, address bits PA[24:30] provide the index to select a set, and bits PA[31:33] select an instruction within a block. The tags consist of physical address bits PA[0:23]. Address translation occurs in parallel with set selection (from PA[24:30]). All of these address ranges are shown for 36-bit physical addressing (enabled when HIDO[XAEN] = 1). When 32-bit addressing is used (HIDO[XAEN] = 0), all of these physical address bits are shifted down by 4, and the tags consist of physical address bits PA[0:19].

The instruction cache is also non-blocking in that it can be accessed internally while a fill for a miss is pending (allowing hits under misses). In addition, subsequent misses can also be sent to the core memory subsystem before the original miss is serviced (allowing misses under misses). When a miss is actually updating the cache, subsequent accesses are blocked for one cycle, but the instruction that missed can be forwarded to the instruction unit simultaneously.

The instruction cache differs from the data cache in that it does not implement a multiple-state cache coherency protocol. A single status bit indicates whether a cache block is valid or invalid. The instruction

cache is not snooped, so if a processor modifies a memory location that may be contained in the instruction cache, **software must ensure that such memory updates are visible to the instruction fetching mechanism.** This can be achieved with the following instruction sequence (using either **dcbst** or **dcbf**):

```

dcbst (or dcbf) |update memory
sync      |wait for update
icbi      |remove (invalidate) copy in instruction cache
sync      |ensure that ICBI invalidate at the instruction cache has completed
isync     |remove copy in own instruction buffer
  
```

These operations are necessary because the processor does not maintain instruction memory coherent with data memory. Software is responsible for enforcing coherency of instruction caches and data memory. Since instruction fetching may bypass the data cache, changes made to items in the data cache may not be reflected in memory until after the instruction fetch completes.

Although not shown in [Figure 3-3](#), the instruction cache has one parity bit/word.

3.3 Memory and Cache Coherency

The primary objective of a coherent memory system is to provide the same image of memory to all devices using the system. Coherency allows synchronization and cooperative use of shared resources. Otherwise, multiple copies of a memory location, some containing stale values, could exist in a system, resulting in errors when the stale values are used. Each potential bus master must follow rules for managing the state of its cache. This section describes the coherency mechanisms of the PowerPC architecture and the cache coherency protocols that the e600 caches support.

Unless specifically noted, the discussion of coherency in this section applies to the L1 data cache and the L2 cache. The instruction cache is not snooped. Instruction cache coherency must be maintained by software. However, the e600 core does support a fast instruction cache invalidate capability as described in [Section 3.4.1.5, “L1 Instruction and Data Cache Flash Invalidation.”](#) Also, [Section 3.4.4.8, “Instruction Cache Block Invalidate \(icbi\),”](#) describes the flushing of self-modifying code from the data cache (and the L2).

3.3.1 Memory/Cache Access Attributes (WIMG Bits)

Some memory characteristics can be set on either a memory management block or page basis by using the WIMG bits in the BAT registers or page table entries (PTE), respectively. These bits allow both uniprocessor and multiprocessor system designs to exploit numerous system-level performance optimizations. The WIMG attributes control the following functionalities:

- Write-through (W bit)
- Caching-inhibited (I bit)
- Memory-coherency-required (M bit)
- Guarded (G bit)

The WIMG attributes are programmed by the operating system for each page and block. The W and I attributes control how the processor performing an access uses its own cache. The M attribute ensures that coherency is maintained for all copies of the addressed memory location. The G attribute prevents loads

and instruction fetches from being performed until they are guaranteed to be required by the sequential execution model.

The WIMG attributes occupy four bits in the BAT registers for block address translation and in the PTEs for page address translation. The WIMG bits are programmed as follows:

- The operating system uses the **mtspr** instruction to program the WIMG bits in the BAT registers for block address translation. The IBAT register pairs do not have a G bit, and all accesses that use the IBAT register pairs are considered not guarded.
- The operating system writes the WIMG bits for each page into the PTEs in system memory as it sets up the page tables.

When an access requires coherency, the processor performing the access must inform the coherency mechanisms throughout the system that the access requires memory coherency. The M attribute determines the kind of access performed on the MPX bus (global or non-global).

3.3.1.1 Coherency Paradoxes and WIMG

Care must be taken with respect to the use of the WIMG bits if coherent memory support is desired. Careless specification of these bits may create situations that present coherency paradoxes to the processor. These coherency paradoxes can occur within a single processor or across several processors. It is important to note that, in the presence of a paradox, the operating system software is responsible for correctness.

In particular, a coherency paradox can occur when the state of these bits is changed without appropriate precautions (such as flushing the pages that correspond to the changed bits from the caches of all processors in the system) or when the address translations of aliased real addresses specify different values for certain WIMG bit values. The e600 core supports aliasing for WIMG = 100x and WIMG = 000x; however, the e600 core does not support aliasing for WIMG = 101x and WIMG = 001x. Specifically, this means that for a given physical address, the e600 core only supports simultaneous memory/cache access attributes for that physical address of write-through, caching-allowed, memory-coherency-not-required (WIMG = 100x) and write-back, caching-allowed, memory-coherency-not-required (WIMG = 000x).

For real addressing mode (that is, for accesses performed with address translation disabled—MSR[IR] = 0 or MSR[DR] = 0 for instruction or data access, respectively), the WIMG bits are automatically generated as 0b0011 (all memory is write-back, caching-allowed, memory-coherency-required, and guarded).

3.3.1.2 Out-of-Order Accesses to Guarded Memory

On the e600 core, instructions are not fetched from guarded memory when instruction translation is enabled (MSR[IR] = 1). If an attempt is made to fetch instructions from guarded memory when MSR[IR] = 1, an ISI interrupt is taken.

The e600 core only fetches instructions out-of-order with respect to other instruction fetches from guarded memory when MSR[IR] = 0 and one of the following conditions applies:

- The instruction is in the instruction cache.
- The instruction resides in the same physical page as an instruction that is required by the execution model.
- The instruction resides in the next sequential physical page from an instruction that is required by the execution model.

Note that the core can have two instruction fetches outstanding at any time.

The e600 core does not perform stores until they are required by the sequential execution model, independent of the setting of the G bit. The only effect of the G bit on stores is that the e600 core guarantees that stores to guarded ($G = 1$) and caching-inhibited ($I = 1$) memory are not store-gathered. (See [Section 3.1.2.3, “Store Gathering/Merging,”](#) for more information on store gathering.)

However, setting the G bit prevents a load from accessing the core interface until it is guaranteed to be required by the sequential execution model. Loads from guarded memory may be accessed out-of-order with respect to other loads from guarded memory if one of the following applies:

- The target location is valid in the data cache.
- The load is guaranteed to be executed. In this case, the entire cache block containing the referenced data may be loaded into the cache.

Note that instruction fetches and loads may also be prevented from accessing the core interface until they are guaranteed to be required by the sequential execution model by setting the speculative access disable bit, `HID0[SPD]`. Also note that setting `HID0[SPD]` does not prevent loads from bypassing stores. See [Section 3.3.3.5, “Enforcing Store Ordering with Respect to Loads,”](#) for more information.

For the e600 core, a guarded load is not allowed to access the core interface until that load is at the bottom of the completion buffer. This means that all prior load accesses to the core interface must have already returned data to the core before the subsequent guarded load is allowed to access the MPX address bus. This prevents the e600 core from pipelining a guarded load with any other type of load on the core interface. Note that this has a large negative effect on load miss bandwidth performance. For this reason, it is not recommended to have guarded loads in code streams that require high system bandwidth utilization.

3.3.2 Coherency Support

The e600 core provides full hardware support for PowerPC cache coherency and ordering instructions (`dcbz`, `dcbi`, `dcbf`, `sync`, `icbi`, and `eieio`) and full hardware implementation of the TLB management instructions (`tlbie`, and `tlbsync`). Snooping, described in [Section 3.7.3, “Snooping of MPX Bus Transactions,”](#) is integral to the core memory subsystem design and operation. The e600 core is self-snooping and can retry its own `stwcx.` broadcasts.

Each 32-byte cache block in the data cache contains two status bits. The e600 core uses these bits to support the coherency protocols and to direct reload operations. The L1 data cache status bits and the conditions that cause them to be set or cleared are defined in [Table 3-1](#). Note that different status bits are used in the L2 cache, as shown in [Table 3-2](#).

Table 3-1. L1 Data Cache Status Bits

MESI [0–2]	Name	Meaning	Set Conditions	Clear Conditions
110	Modified (M)	The cache block is modified with respect to the core interface	<ul style="list-style-type: none"> Store miss reload from MPX bus or L2 cache Write-back store hit on \negS 	Snoop hit
100	Exclusive (E)	The cache block is valid	Reload from MPX bus or L2 cache	<ul style="list-style-type: none"> dcbi, dcbf, and dcbst hit Write-back store hit to S (see Section 3.5.5, “Store Hit to a Data Cache Block Marked Shared”) Snoop clean hit Snoop invalidate hit
101	Shared (S)	The cache block is shared with other processors and is read-only	<ul style="list-style-type: none"> Load miss reload from MPX bus with shared response Load miss reload from L2 cache with L2 cache status = S 	None
0xx	Invalid (I)	—	—	—

Table 3-2. L2 Cache Status Bits

MESI [0–1]	Name	Meaning	Set Conditions	Clear Conditions
11	Modified (M)	The cache block is modified with respect to the core interface	<ul style="list-style-type: none"> Store miss reload from MPX bus or L2 cache Write-back store hit on \negS 	Snoop hit
10	Exclusive (E)	The cache block is valid	Reload from MPX bus or L2 cache	<ul style="list-style-type: none"> dcbi, dcbf, and dcbst hit Write-back store hit to S (see Section 3.5.5, “Store Hit to a Data Cache Block Marked Shared”) Snoop clean hit Snoop invalidate hit
01	Shared (S)	The cache block is shared with other processors and is read-only	<ul style="list-style-type: none"> Load miss reload from MPX bus with shared response Load miss reload from L2 cache with L2 cache status = S 	None
00	Invalid (I)	—	—	—

Every data cache block state is defined by its MESI status bits. Note that in a multiprocessor system, a cache line can exist in the exclusive state in at most one L1 data cache at any one time.

3.3.2.1 Coherency Between L1 and L2 Caches

The e600 core allows for the L1 data cache and the L2 cache to have different coherency status for the same cache block. For example, a cache block in the L2 cache is allowed to be shared when the same block

in the L1 cache is exclusive or modified. The true coherency state of a cache block within the e600 core is determined by analyzing all levels of the cache hierarchy.

3.3.2.1.1 Cache Closer to Core with Modified Data

A cache block can be in the shared, exclusive, or modified state in the L2 while a cache closer to the processor core has the block in the modified state. In this case the cache closer to the core may have newer data. So by definition, if a cache block is in the shared, exclusive, or modified state in the L1 or L2, it has the newest data if no cache closer to the processor core has the block in the modified state.

If a cache block is in the modified state in the L2 and that block is modified in the L1, the L2 may cast out out-of-date data to memory. In this case, the newest data still exists in the L1.

3.3.2.1.2 Transient Data and the L2 Cache

The allowance of different cache states between the L1 and L2 caches eliminates the need to allocate or update the state in the L2 when a transient (**dststt** or **stvx1**) store occurs to a block that is marked as shared in the L2 . In this case, the LLQ treats the L2 block as invalid for stores.

3.3.2.2 Snoop Response

Table 3-3 describes the snoop responses used by the e600 core and defines the symbols used in Section 3.3.2.5, “MESI State Transitions.” Chapter 8, “Core Interface,” does not include detailed signal timing and bus protocol information.

Table 3-3. Snoop Response Summary

Snoop Response on the MPX Bus	State Transition Diagram Symbol	Description
No response	— (No symbol)	The processor does not contain any memory at the snooped address or the coherency protocol does not require a response. The snoop has been fully serviced and no internal pipeline collisions occurred that would require a busy response.
Shared indicated	S	The processor contains data from the snooped address or a reservation on the snooped address.
Retry asserted	A	The processor cannot service the snoop due to an internal pipeline collision (busy). The same address tenure must be rerun at a later time.
Retry followed by bus request	AS	The processor contains a modified copy of data from the snooped address and is prepared to perform a window-of-opportunity (W) snoop push. The same address tenure must be rerun at a later time.
Hit indicated	H1	The processor contains a modified copy of data from the snooped address and is prepared to perform cache-to-cache or window-of-opportunity (C or W) intervention.

3.3.2.3 Intervention

Table 3-4 briefly describes the intervention types used by the e600 core.

Table 3-4. Snoop Intervention Summary

Intervention Type	State Transition Diagram Symbol	Description
No intervention	— (No symbol)	The core does not contain any memory at the snooped address or the coherency protocol does not require intervention.
Window-of-opportunity	W	<p>Window-of-opportunity snoop push for hits on modified data. The core performs a write-with-kill, snoop-push transaction in the next address tenure using what is referred to as a window of opportunity to initiate the snoop push operation.</p> <p>When a master uses the window of opportunity, it performs a snoop push (write-with-kill) to the most previous snoop address (unless the master still has a write-with-kill pending due to a previous window-of-opportunity request that is not yet satisfied). The e600 core always presents a cache-block aligned address (that is, A[31:35] = 0b0_0000) for every window-of-opportunity snoop push.</p>
Cache-to-cache	C	Cache-to-cache intervention for hits on modified data. The core has queued up a data-only write transaction to provide data to the snooping master (cache-to-cache intervention). If another master retries the transaction coincident with assertion of a hit, the e600 core cancels the queued-up data-only transaction but does not attempt to perform a window-of-opportunity snoop push. The cache block state is already changed to the new state due to the snoop. Thus, the intervening core (the one that indicated a hit) does not contain the cache block in a state suitable for intervention when the retried snoop transaction is rerun on the MPX bus. However, it can perform a window-of-opportunity snoop push when the retried snoop transaction is rerun.

3.3.2.4 Simplified Transaction Types

For the purposes of snooping MPX bus transactions, the e600 core treats related (but distinct) transaction types as a single simplified transaction type. Table 3-5 defines the mapping of simplified transaction types to actual transaction types.

Table 3-5. Simplified Transaction Types

Simplified Transaction Type	Actual Transaction Type
Read	Read Read-atomic
RWITM	RWITM (read-with-intent-to-modify) RWITM-atomic RCLAIM (read-claim)
RWNITC	RWNITC (read-with-no-intent-to-cache)—Acts like a read transaction for snoop response purposes; acts like a clean transaction for MESI state change purposes.
Write	Write-with-flush Write-with-flush-atomic
Flush	Flush

Table 3-5. Simplified Transaction Types (continued)

Simplified Transaction Type	Actual Transaction Type
Clean	Clean
Kill	Kill Write-with-kill
Reskill (Used for reservation snooping only)	RWITM RWITM-atomic RCLAIM Write-with-flush Write-with-flush-atomic Kill Write-with-kill

In the following state transition diagrams, RWNITC is not explicitly shown. For state transitions (for example, modified to exclusive), the e600 core treats RWNITC like a clean operation. For intervention purposes (for example a W or C intervention as defined in [Table 3-4](#)), the core treats RWNITC like a read operation.

3.3.2.5 MESI State Transitions

The state diagrams in this section use symbols on the transition lines for snoop response and intervention type. For example, H1-C denotes a hit-indicated snoop response and a cache-to-cache intervention type. See [Table 3-3](#) and [Table 3-4](#) for the symbols used in the state diagrams.

3.3.2.5.1 MESI Protocol with Data Intervention Enabled

The following state diagrams ([Figure 3-4](#), [Figure 3-5](#), [Figure 3-6](#), [Figure 3-7](#), and [Figure 3-8](#)) show the MESI state transitions when the e600 core is configured for modified data intervention enabled (MSSCR0[EIDIS] = 0).

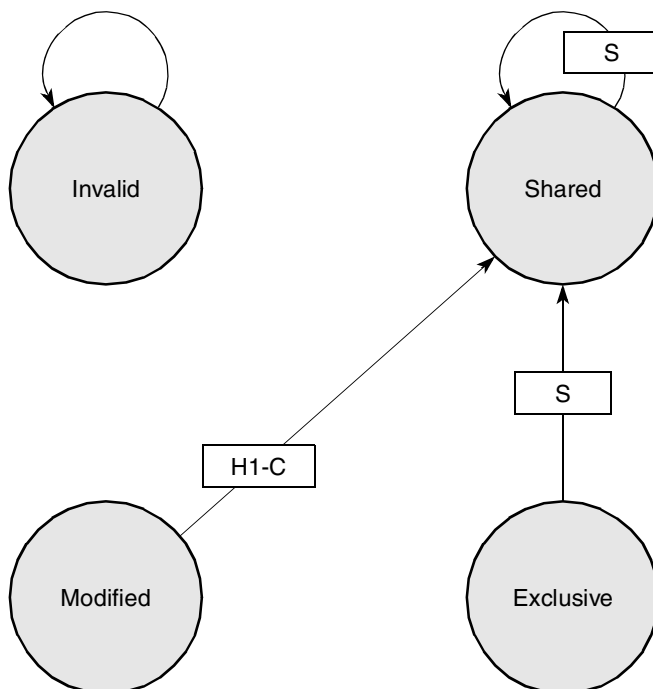


Figure 3-4. Read Transaction—MSSCR0[EIDIS] = 0

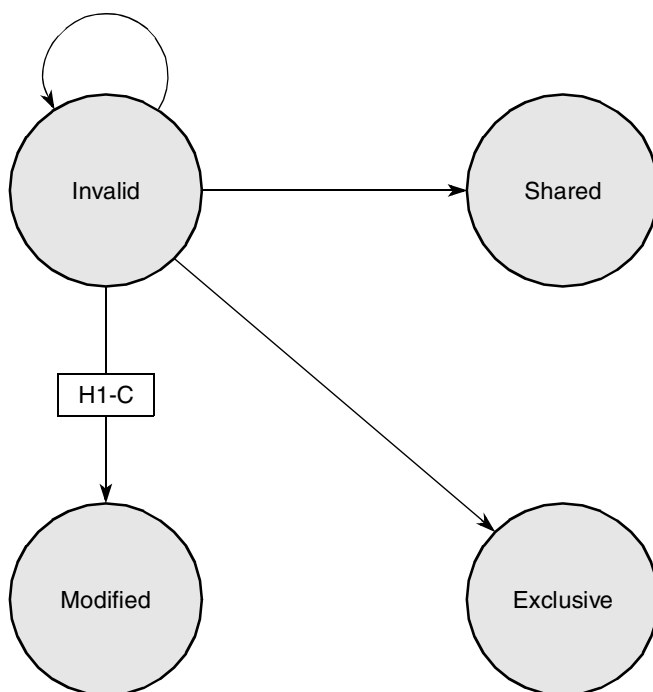


Figure 3-5. RWITM and Flush Transactions—MSSCR0[EIDIS] = 0

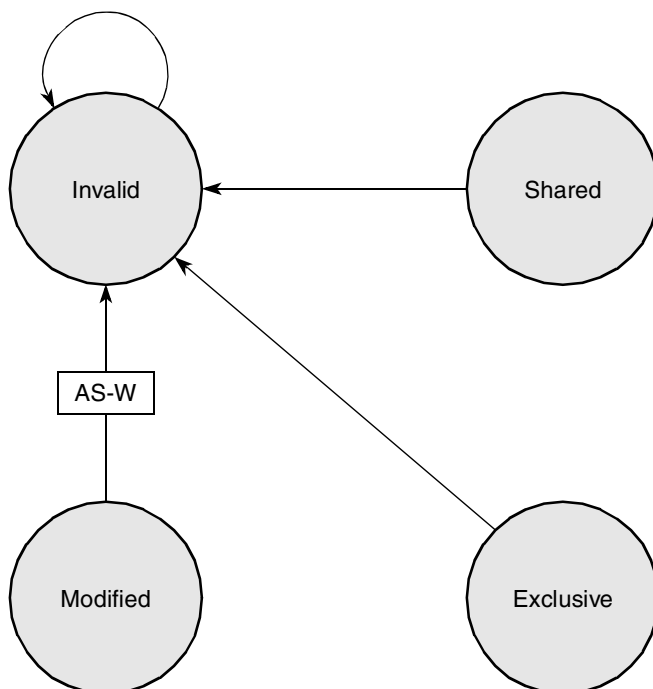


Figure 3-6. Write Transaction—MSSCR0[EIDIS] = 0

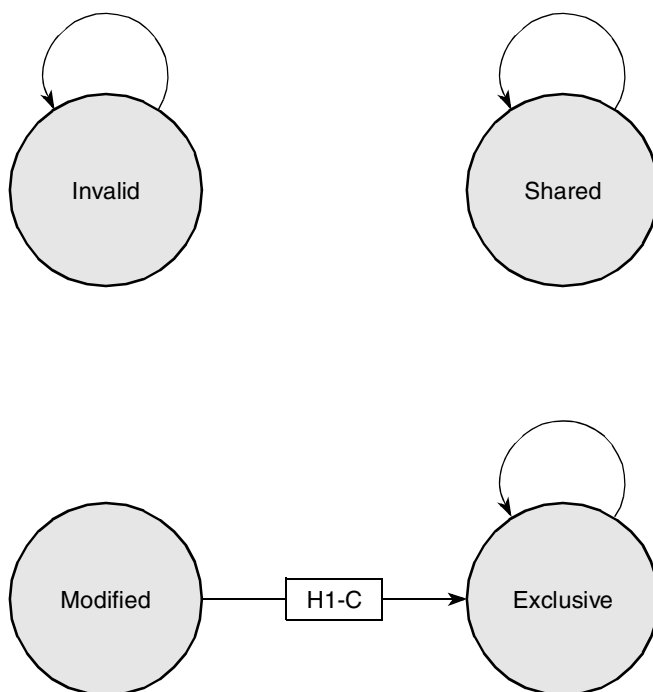
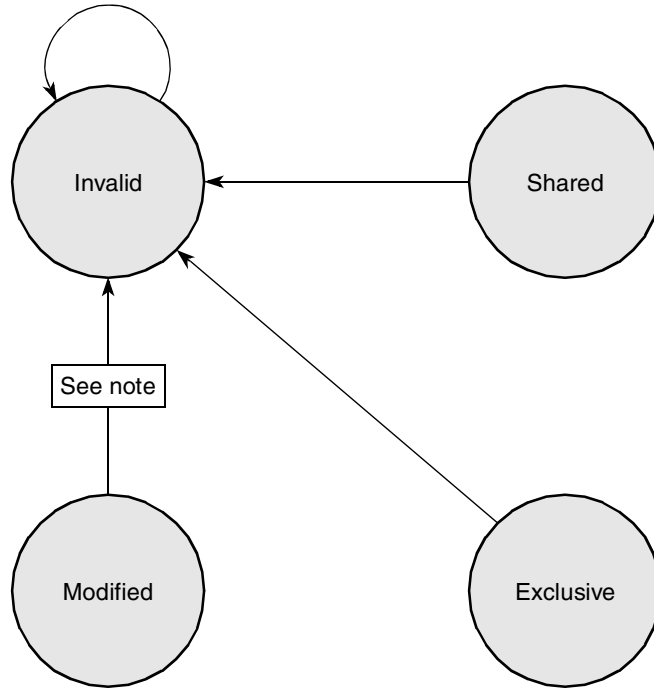


Figure 3-7. Clean Transaction—MSSCR0[EIDIS] = 0



Note: If another master retries, the e600 core performs a window-of-opportunity style push. Otherwise, there is no intervention.

Figure 3-8. Kill Transaction—MSSCR0[EIDIS] = 0

3.3.2.5.2 MESI Protocol (with Intervention Disabled)

The following state diagrams (Figure 3-9, Figure 3-10, Figure 3-11, and Figure 3-12) show the MESI state transitions when the e600 core is configured for when hit intervention is disabled (MSSCR0[EIDIS] = 1).

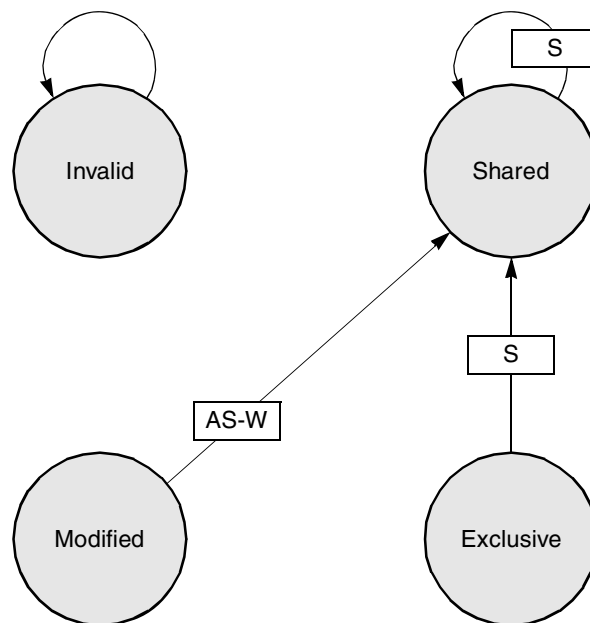
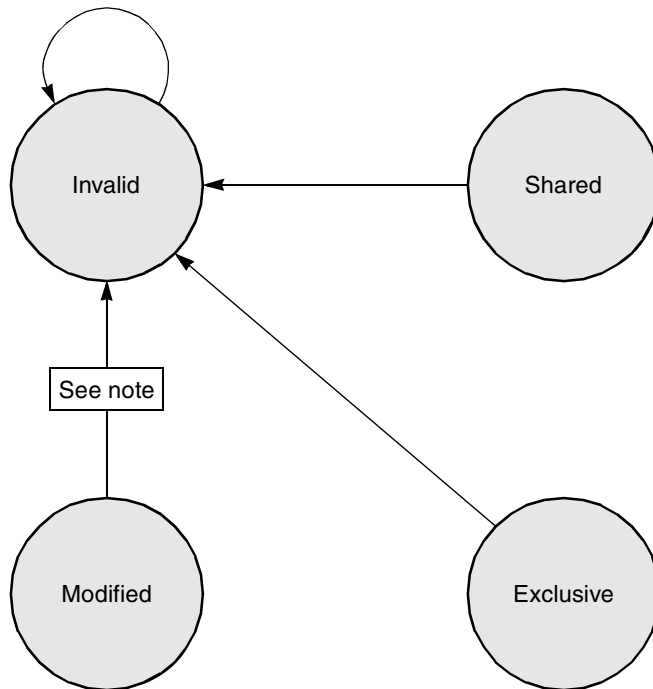


Figure 3-9. Read Transaction—MSSCR0[EIDIS] = 1



Note: If another master retries, the e600 core performs a window-of-opportunity style push. Otherwise, there is no intervention.

Figure 3-12. Kill Transaction—MSSCR0[EIDIS] = 1

3.3.2.6 Reservation Snooping

The e600 core snoops all transactions against the contents of the reservation address register independent of the cache snooping. The following state diagrams (Figure 3-13, Figure 3-14, and Figure 3-15) show the response to those snoops.

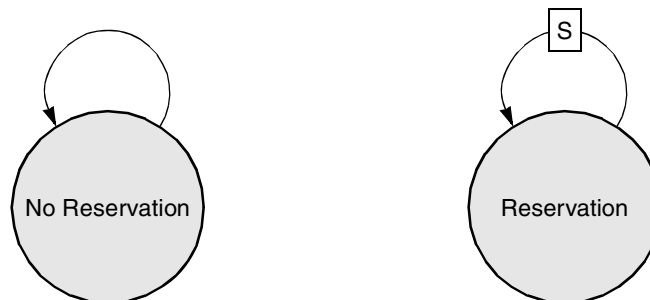


Figure 3-13. Read Transaction Snoop Hit on the Reservation Address Register

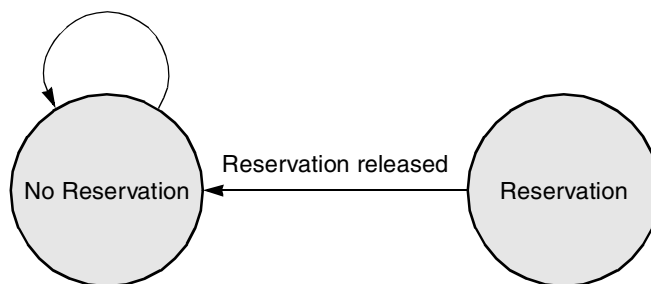


Figure 3-14. Reskill Transaction Snoop Hit on the Reservation Address Register

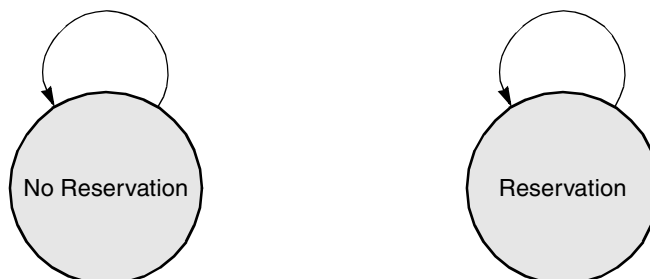


Figure 3-15. Other Transaction Snoop Hit on the Reservation Address Register

3.3.3 Load/Store Operations and Architecture Implications

Load and store operations are assumed to be weakly ordered on the e600 core. The load/store unit (LSU) can perform load operations that occur later in the program ahead of store operations, even when the data cache is disabled (see [Section 3.3.3.2, “Sequential Consistency of Memory Accesses”](#)).

The core does not provide support for direct-store segments. Operations attempting to access a direct-store segment cause a DSI interrupt. For additional information about DSI interrupts, refer to [Section 4.6.3, “DSI Interrupt \(0x00300\).”](#)

3.3.3.1 Performed Loads and Store

The PowerPC architecture defines a performed load operation as one that has the addressed memory location bound to the target register of the load instruction. The architecture defines a performed store operation as one where the stored value is the value that any other processor will receive when executing a load operation (that is, of course, until it is changed again). With respect to the e600 core, caching-allowed (WIMG = x0xx) loads and caching-allowed, write-back (WIMG = 00xx) stores are performed when they have arbitrated to address the cache block in the L1 data cache, the L2 cache, or the core interface. Note that loads are considered performed at the L1 data cache or L2 cache only if the respective cache contains a valid copy of that address. Write-back stores are considered performed at the L1 data cache or L2 cache only if the respective cache contains a valid, non-shared copy of that address. Caching-inhibited (WIMG = x1xx) loads and stores, and write-through (WIMG = 10xx) stores are considered performed when they have been successfully presented to the MPX bus. A set of rules for load and store ordering using the WIMG bits in the BAT registers or page table entries (PTE) in the e600 core is listed in [Table 3-6](#).

Table 3-6. Load and Store Ordering with WIMG Bit Settings

W	I	M	G	Order ^{1, 2}
<i>n</i>	1	<i>n</i>	1	Stores are ordered with respect to other stores. Loads are ordered with respect to other loads. A store followed by a load requires an eiio instruction in between the store and load.
1	0	<i>n</i>	1	Stores are ordered with respect to other stores. Loads are ordered with respect to other loads. A store followed by a load requires a sync instruction in between the store and load.
1	<i>n</i>	<i>n</i>	0	Stores are ordered with respect to other stores. A load followed by a load requires a sync instruction in between the loads. A store followed by a load requires a sync instruction in between the store and load.
0	0	1	<i>n</i>	A store followed by a store requires an eiio instruction in between the stores. A load followed by a load requires a sync instruction in between the loads. A store followed by a load requires a sync instruction in between the store and load.
0	0	0	<i>n</i>	A store followed by a store requires an eiio instruction in between the stores. A load followed by a load requires a sync instruction in between the loads. A store followed by a load requires a sync instruction in between the store and load.
0	1	<i>n</i>	0	A store followed by a store requires an eiio instruction in between the stores. A load followed by a load requires a sync instruction in between the loads. A store followed by a load requires a sync instruction in between the store and load.

¹ Any load followed by any store is always ordered for the e600 core.

² A **sync** instruction will cover the synchronization cases that require an **eiio** instruction. However, an **eiio** instruction will not cover all the synchronization cases that require a **sync** instruction.

3.3.3.2 Sequential Consistency of Memory Accesses

The PowerPC architecture requires that all memory operations executed by a single processor be sequentially consistent with respect to that processor. This means that all memory accesses appear to be executed in program order with respect to interrupts and data dependencies.

The e600 core achieves sequential consistency by operating a single data pipeline to the cache/MMU. All memory accesses are presented to the MMU in exact program order and therefore interrupts are taken in program order. Loads are allowed to bypass stores after interrupt checking has been performed for the store, but data dependency checking is handled in the load/store unit so that a load does not bypass a store with an address match. Newer caching-allowed loads can bypass older caching-allowed loads only if the two loads are to different 32-byte address granules. Newer caching-allowed write-back stores can bypass older caching-allowed write-back stores if they do not store to overlapping bytes of data.

Note that although memory accesses that miss in the L1 cache are forwarded to the load/store unit load queue for future arbitration for the L2 cache (and possibly the MPX bus), all potential synchronous interrupts have been resolved before the L1 cache access. In addition, although subsequent memory accesses can address the L1 cache, full coherency checking between the L1 cache and the load/store unit load and store queues is provided to avoid dependency conflicts.

3.3.3.3 Load Ordering with Respect to Other Loads

The PowerPC architecture guarantees that the following loads are not re-ordered with respect to other similar loads:

- Caching-inhibited ($I = 1$) and guarded ($G = 1$) loads

The e600 core guarantees that the following loads are not re-ordered with respect to other similar loads:

- Caching-inhibited ($I = 1$) loads when $HID0[SPD] = 1$

Note that when address translation is disabled (real addressing mode), the default WIMG bits cause the I bit to be cleared (accesses are assumed to be caching-allowed), and thus the load accesses are weakly ordered with respect to each other. Refer to [Section 5.2, “Real Addressing Mode,”](#) for a description of the WIMG bits when address translation is disabled.

3.3.3.4 Store Ordering with Respect to Other Stores

The PowerPC architecture also guarantees that the following stores are not re-ordered with respect to other similar stores:

- Caching-inhibited ($I = 1$) stores

Additionally, the e600 core also guarantees that the following stores are not re-ordered with respect to other similar stores:

- Write-through ($W = 1$) stores

Otherwise, stores on the e600 core are weakly ordered with respect to other stores.

3.3.3.5 Enforcing Store Ordering with Respect to Loads

The PowerPC architecture specifies that an **eieio** instruction must be used to ensure sequential ordering of loads with stores.

The e600 core guarantees that any load followed by any store is performed in order (with respect to each other). The reverse, however, is not guaranteed. An **eieio** instruction must be inserted between a store followed by a load to ensure sequential ordering between that store and that load. Also note that setting $HID0[SPD]$ does not prevent loads from bypassing stores.

If store gathering is enabled (through $HID0[SGE]$), the **eieio** instruction may also be used to keep stores from being gathered. If an **eieio** instruction is detected in the store queues, store gathering is not performed. If $HID1[SYNCBE] = 1$, the **eieio** instruction also causes an MPX bus broadcast operation, which may be used to prevent devices or peripheral blocks external to the e600 core, such as bus bridge chips, from gathering stores. See [Section 3.1.2.3, “Store Gathering/Merging,”](#) for more information on store gathering.

3.3.3.6 Atomic Memory References

The PowerPC architecture defines the Load Word and Reserve Indexed (**lwarx**) and the Store Word Conditional Indexed (**stwcx.**) instructions to provide an atomic update function for a single, aligned word of memory. These instructions can be used to develop a rich set of multiprocessor synchronization primitives. Note that atomic memory references constructed using **lwarx/stwcx.** instructions depend on

the presence of a coherent memory system for correct operation. These instructions should not be expected to provide atomic access to noncoherent memory. For detailed information on these instructions, refer to [Chapter 2, “Registers,”](#) in this book and Chapter 8, “Instruction Set,” in *The Programming Environments Manual*.

The **lwarx** instruction performs a load word from memory operation and creates a reservation for the 32-byte section of memory that contains the accessed word. The reservation granularity is 32 bytes. The **lwarx** instruction makes a non-specific reservation with respect to the executing processor and a specific reservation with respect to other masters. This means that any subsequent **stwcx.** executed by the same processor, regardless of address, cancels the reservation. Also, any bus write or invalidate operation from another processor to an address that matches the reservation address cancels the reservation.

The **stwcx.** instruction does not check the reservation for a matching address. The **stwcx.** instruction is only required to determine whether a reservation exists. The **stwcx.** instruction performs a store word operation only if the reservation exists. If the reservation has been cancelled for any reason, then the **stwcx.** instruction fails and clears the CR0[EQ] bit in the condition register. The architectural intent is to follow the **lwarx/stwcx.** instruction pair with a conditional branch which checks to see whether the **stwcx.** instruction failed.

Executing an **lwarx** or **stwcx.** instruction to areas marked write-through or cache-inhibited causes a DSI interrupt. Additionally, executing an **lwarx** or **stwcx.** instruction when the L1 data cache is disabled or it is enabled and locked causes a DSI interrupt.

If the corresponding page table entry is marked caching-allowed (WIMG = x0xx) and an **lwarx** access misses in the cache, the e600 core performs a cache block fill. All MPX bus operations that are a direct result of either an **lwarx** instruction or an **stwcx.** instruction are placed on the MPX bus with a special encoding. Note that this does not force all **lwarx** instructions to generate MPX bus transactions, but rather provides a means for identifying when an **lwarx** instruction does generate a MPX bus transaction.

The e600 core snoops its own RWITM-atomic transactions to check the state of the reservation bit. If the reservation is set, the RWITM-atomic transaction succeeds. Otherwise, the core internally retries it and the transaction is re-sent as a read transaction.

3.4 L1 Cache Control

The e600 core L1 caches are controlled by programming specific bits in the HID0, ICTRL, and LDSTCR special-purpose registers and by issuing dedicated cache control instructions. [Section 3.4.1, “Cache Control Parameters in HID0,”](#) describes the HID0 cache control bits, [Section 3.4.2, “Data Cache Way Locking Setting in LDSTCR,”](#) describes the data cache way locking feature and [Section 3.4.3, “Cache Control Parameters in ICTRL,”](#) describes the L1 cache parity checking features and the instruction cache way locking. Note that the ICTC register also affects the instruction cache operation and it is described in [Section 9.7, “Instruction Cache Throttling.”](#)

Also, [Section 2.1.6.1, “Hardware Implementation-Dependent Register 0 \(HID0\),”](#) [Section 2.1.6.3, “Memory Subsystem Control Register \(MSSCR0\),”](#) and [Section 2.1.6.5.16, “Load/Store Control Register \(LDSTCR\),”](#) provide detailed information on the bit settings for these registers.

Finally, [Section 3.4.4, “Cache Control Instructions,”](#) describes the cache control instructions.

See [Section 3.6.3, “L2 Cache Control,”](#) for information on the L2 cache control functions.

3.4.1 Cache Control Parameters in HID0

The HID0 special-purpose register contains several bits that invalidate, disable, and lock the instruction and data caches. The following sections describe these L1 cache control facilities.

3.4.1.1 Enabling and Disabling the Data Cache

The data cache is enabled or disabled with the data cache enable bit, HID0[DCE]. HID0[DCE] is cleared on power-up, disabling the data cache. Snooping is not performed when the data cache is disabled.

When the data cache is in the disabled state (HID0[DCE] = 0), the cache tag status bits are ignored, and all data accesses are propagated to the MPX bus as single- or double-beat cache-inhibited (\overline{ci} asserted) transactions, depending on the size of the access. Thus, they are ignored by the L2 cache, independent of the state of the L2. Note that disabling the data cache does not affect the address translation logic; translation for data accesses is controlled by MSR[DR].

The setting of the DCE bit must be preceded by a **sync** instruction to prevent the cache from being enabled or disabled in the middle of a data access. In addition, the cache must be globally flushed before it is disabled to prevent coherency problems when it is re-enabled. See [Section 3.5.8, “L1 Cache Invalidation and Flushing,”](#) for more information on the flushing of the data cache.

The **dcbz** instruction causes an alignment interrupt when the access is to a cache-inhibited or write-through area of memory. Thus a **dcbz** causes an alignment interrupt for the cases when the data cache is disabled (HID0[DCE] = 0), or when the data cache is completely locked (LDSTCR[DCWL] = 0xFF or HID0[DLOCK] = 1). The touch load (**dcbt** and **dcbst**) instructions are no-ops when the data cache is disabled; however, address translation is still performed for these instructions. Other cache instructions (**dcbf**, **dcbst**, and **dcbi**) do not affect the data cache when it is disabled.

Note that if the L1 data cache is disabled, the L2 cache may be enabled, but it ignores all data accesses. The L2 cache is enabled or disabled with L2CR[L2E].

3.4.1.2 Data Cache Locking with DLOCK

The entire contents of the data cache can be locked by setting the data cache lock bit, HID0[DLOCK]. No new tags are allocated for a locked data cache. Snoop hits, store hits (to mark the line modified), and **dcbf**, **dcbi**, and **dcbst** instructions are the only operations that can cause a tag state change in a locked data cache. If all ways of the data cache are locked, all stores are sent to the core memory subsystem as cacheable but write-through (as if W = 1). Accesses caused by the **dcbz** instruction when the data cache is completely locked take an alignment interrupt as described in [Section 3.4.1.1, “Enabling and Disabling the Data Cache.”](#) However, accesses caused by the **dcba** instruction when the data cache is completely locked are treated as no-ops.

The setting of the DLOCK bit must be preceded by a **dssall/sync** instruction pair and followed by a **sync** instruction to prevent the data cache from being locked during a data access. Also, the data cache should be already enabled when setting DLOCK.

The e600 core treats a load hit to a locked data cache the same as a load hit to an unlocked data cache. That is, the data cache services the load with the requested data. However, a load that misses in a locked data cache is passed to the LMQ and propagates to the L2 cache or MPX bus as a caching-allowed, 32-byte

burst read. In this case, the data is forwarded to the requesting execution unit when it returns, but it is not loaded into the data cache.

The e600 core treats snoop hits to a locked data cache the same as snoop hits to an unlocked data cache. However, any cache block invalidated by a snoop hit remains invalid and is not reallocated until the cache is unlocked.

One to eight ways of the data cache can be locked by setting bits in LDSTCR. See [Section 3.4.2, “Data Cache Way Locking Setting in LDSTCR,”](#) for more information on way locking of the data cache.

3.4.1.3 Enabling and Disabling the Instruction Cache

The instruction cache may be enabled or disabled through the use of the instruction cache enable bit, HID0[ICE]. HID0[ICE] is cleared on power-up, disabling the instruction cache. The setting of the ICE bit must be preceded by an **isync** instruction to prevent the cache from being enabled or disabled in the middle of an instruction fetch. Furthermore, the setting of the ICE bit must be followed by an **isync** instruction in order for the setting to take effect. The **icbi** instruction is not affected by disabling the instruction cache. For further details on synchronization see [Section 2.3.2.4.1, “Context Synchronization.”](#)

When the instruction cache is in the disabled state (HID0[ICE] = 0), the cache tag status bits are ignored, and all instruction fetches are forwarded to the L2 cache and the core memory subsystem with the cacheability attribute determined by the WIMG bits. When the instructions are returned, they are forwarded to the instruction unit, but are not loaded into the instruction cache. Note that the \overline{ci} signal always reflects the state of the caching-inhibited memory/cache access attribute (the I bit) for instruction accesses independent of the state of HID0[ICE]. Also note that disabling the instruction cache does not affect the address translation logic; translation for instruction accesses is controlled by MSR[IR].

3.4.1.4 Instruction Cache Locking with ILOCK

The contents of the instruction cache can be locked by setting the instruction cache lock bit, HID0[ILOCK]. A completely locked instruction cache has no new tag allocations. **icbi** instructions are the only operations that can cause a tag state change in a locked instruction cache. The setting of the ILOCK bit must be preceded by an **isync** instruction to prevent the instruction cache from being locked during an instruction fetch.

An instruction fetch that hits in a locked instruction cache is serviced by the cache. An instruction fetch that misses in a completely locked instruction cache is propagated to the L2 and MPX bus as a 32-byte burst read. When the instructions are returned, they are forwarded to the instruction unit but are not loaded into the instruction cache.

Note that the \overline{ci} signal always reflects the state of the caching-inhibited memory/cache access attribute (the I bit) for instruction accesses independent of the state of HID0[ILOCK]. See [Section 3.4.3.1, “Instruction Cache Way Locking,”](#) for information on the locking of one to eight ways of the instruction cache.

3.4.1.5 L1 Instruction and Data Cache Flash Invalidation

The HID0[ICFI] and HID0[DCFI] bits of the e600 core cause a flash invalidation of the instruction and data caches, respectively. Each cache can be flash invalidated independently. Note that HID0[ICFI] and

HID0[DCFI] must not both be set with the same **mtspr** instruction, due to the synchronization requirements described in [Section 2.3.2.4.1, “Context Synchronization.”](#)

A reset operation does not invalidate the caches. Therefore, software must flash invalidate the instruction cache with the same **mtspr** to HID0 instruction that enables the instruction cache, and it must flash invalidate the data cache with the same **mtspr** to HID0 instruction that enables the data cache. When either HID0[ICFI] or HID0[DCFI] is set by software, the corresponding cache invalidate bit is cleared automatically in the following clock cycle. Note that there is no broadcast of a flash invalidate operation. An **isync** must precede the setting of the HID0[ICFI] in order for the setting to take effect.

Individual instruction cache blocks can be invalidated using the **icbi** instruction and individual data cache blocks can be invalidated using the **dcbi** instruction. See [Section 3.4.4.8, “Instruction Cache Block Invalidate \(icbi\),”](#) and [Section 3.4.4.7, “Data Cache Block Invalidate \(dcbi\),”](#) for more information about the **icbi** and **dcbi** instructions, respectively.

3.4.2 Data Cache Way Locking Setting in LDSTCR

The 8-bit DCWL parameter in LDSTCR controls the locking of from one to eight ways of the data cache. Each bit in DCWL corresponds to a way of the data cache. Bit 24 corresponds to way 0, and bit 31 corresponds to way 7. Setting a bit in DCWL locks the corresponding way in the cache. The e600 core treats a load hit to a locked way in the data cache the same as a load hit to an unlocked data cache. That is, the data cache services the load with the requested data. Also, snoop hits and store hits to a locked way in the data cache also operate the same as a hit to an unlocked cache. However, locked ways are never selected for replacement.

Setting all eight bits is equivalent to setting the HID0[DLOCK] bit. See [Section 3.4.1.2, “Data Cache Locking with DLOCK,”](#) for more information. See [Section 3.5.7.4, “Cache Locking and PLRU,”](#) for more information on PLRU precautions with way locking.

3.4.3 Cache Control Parameters in ICTRL

The ICTRL controls instruction and data cache parity checking and error reporting and enables instruction cache way locking

3.4.3.1 Instruction Cache Way Locking

Similar to the DCWL parameter in LDSTCR, the 8-bit ICWL parameter in ICTRL controls the locking of from one to eight ways of the instruction cache. Each bit in ICWL corresponds to a way of the instruction cache. Bit 24 corresponds to way 0, and bit 31 corresponds to way 7. Setting a bit in ICWL locks the corresponding way in the cache. The e600 core treats a hit to a locked way in the instruction cache the same as a hit to an unlocked instruction cache. That is, the cache services the fetch with the requested instructions. However, on a miss, locked ways are never selected for replacement.

Setting all 8 bits in ICWL is equivalent to setting the HID0[ILOCK] bit. See [Section 3.4.1.4, “Instruction Cache Locking with ILOCK,”](#) for more information. See [Section 3.5.7.4, “Cache Locking and PLRU,”](#) for more information on PLRU precautions with way locking.

3.4.3.2 Enabling Instruction Cache Parity Checking

Instruction cache parity checking is enabled with ICTRL[EICP]. When this bit is set, the parity of all instructions fetched from the L1 cache is checked. See [Section 3.4.3.3, “Instruction and Data Cache Parity Error Reporting,”](#) for information on the reporting of L1 cache parity errors.

3.4.3.3 Instruction and Data Cache Parity Error Reporting

Instruction and data cache parity errors are reported through the machine check interrupt mechanism if ICTRL[EICE] and ICTRL[EDCE] are set, respectively. In order for an instruction cache parity error to be reported, ICTRL[EICP] must also be set. Note that data parity checking is always enabled. When ICTRL[EICE] and ICTRL[EDCE] are cleared, instruction and data cache parity errors are masked. Note that when parity checking and reporting is enabled, parity errors can be reported (causing a machine check) for speculative fetches that result in a parity error, even if the access is never required.

3.4.4 Cache Control Instructions

The PowerPC architecture defines instructions for controlling both the instruction and data caches (when they exist). The cache control instructions: **dcbt**, **dcbst**, **dcbz**, **dcbst**, **dcbf**, **dcba**, **dcbi**, and **icbi**—are intended for the management of the L1 caches. The e600 core interprets the cache control instructions as if they pertain only to its own L1 caches. These instructions are not intended for managing other caches in the system (except to the extent necessary to maintain coherency).

The e600 core snoops all global (\overline{gbl} asserted) cache control instruction broadcasts. The **dcbst**, **dcbf**, and **dcbi** instructions cause a broadcast on the MPX bus (when $M = 1$) to maintain coherency. When $M = 0$, the broadcast of those instructions (and **icbi**, **tlbie**, and **tlbsync**) is controlled by the HID1[ABE] parameter. Therefore, HID1[ABE] must be set in multiprocessor systems.

The e600 core treats any cache control instruction directed to a direct-store segment ($SR[T] = 1$) as a no-op.

3.4.4.1 Data Cache Block Touch (dcbt)

The Data Cache Block Touch (**dcbt**) instruction provides potential system performance improvement through the use of a software-initiated prefetch hint. Note that implementations that support the PowerPC architecture are not required to take any action based on the execution of these instructions, but they may choose to prefetch the cache block corresponding to the effective address into their cache.

If the effective address of a **dcbt** instruction is directed to a direct-store segment ($SR[T] = 1$)x, or if HID0[NOPTI] = 1, the e600 core treats the instruction as a no-op without translation.

If the effective address of a **dcbt** instruction is not directed to a direct-store segment [$T = 0$] and HID0[NOPTI] = 0, the effective address is computed, translated, and checked for protection violations as defined in the PowerPC architecture. The **dcbt** instruction is treated as a load to the addressed byte with respect to address translation and protection. Note, however that a table search operation is never initiated for a **dcbt** instruction.

Additionally, the e600 core treats the **dcbt** instruction as a no-op if any of the following occur:

- A valid address translation is not found in the BAT or TLB
- Load accesses are not permitted to the addressed page (protection violation)
- The BAT or PTE is marked caching-inhibited ($I = 1$)
- The BAT or PTE is marked guarded ($G = 1$) and the **dcbt** instruction is not at the bottom of the completion queue
- The data cache is locked or disabled

If none of the conditions for a no-op are met, the e600 core checks if the addressed cache block is in the L1 data cache. If the cache block is not in the L1 data cache, the core checks if the addressed cache block is in the L2 cache. If the cache block is not in the L2 cache, the e600 core initiates a burst read (with no intent to modify) on the MPX bus.

The data brought into the cache as a result of this instruction is validated in the same manner that a load instruction would be (that is, it is marked as exclusive or shared). Note that the successful execution of the **dcbt** instruction affects the state of the TLB and cache LRU bits as defined by the PLRU algorithm (see [Section 3.5.7, “L1 Cache Block Replacement Selection”](#)).

3.4.4.2 Data Cache Block Touch for Store (**dcbtst**)

The Data Cache Block Touch for Store (**dcbtst**) instruction behaves similarly to the **dcbt** instruction except that it attempts to gain ownership of the line by sending a request on the MPX bus if the data is not found in the L1 or L2 caches in the exclusive or exclusive-modified state. Additionally, there are the following differences from **dcbt**:

- If the target address of a **dcbtst** instruction is marked write-through ($W = 1$), the instruction is treated as a no-op.
- If the **dcbtst** hits in the L1 data cache, the state of the block is not changed.
- If the **dcbtst** misses in the L1 data cache, but hits in the L2 cache as exclusive modified, the data is brought into the L1 data cache and is marked as exclusive.
- If the **dcbtst** misses in the L1 data cache, but hits in the L2 cache as shared, it is treated as a miss.
- If the **dcbtst** misses in both the L1 data cache and the L2 cache, the cache block fill request is signaled on the MPX bus as a read-claim and the data is marked exclusive when it is brought into the L1 data cache from the MPX bus if the response on the MPX bus is not shared.

From a programming point of view, it can be advantageous to use **dcbtst** instructions on the e600 core if multiple line misses may occur that would otherwise be caused by store instructions. This is because the e600 core supports only two outstanding store misses (from CSQ0 and CSQ1), but **dcbtst** line misses are handled in the five-entry LMQ (so up to five **dcbtst** misses could be handled simultaneously).

If **dcbtst** (or **dstst**) is being used to prefetch a 32-byte coherency granule that will eventually be fully consumed by 32-byte's worth of stores (that is, two back-to-back AltiVec **stvx** instructions), the inclusion of touch-for-store may reduce performance if the MPX bus is bandwidth-limited. This is because a touch-for-store must perform both a 32-byte coherency operation on the address bus (two or more bus cycles) and a 32-byte data transfer (four or more bus cycles). On the other hand, caching-allowed, write-back stores that merge to 32-bytes only require a 32-byte coherency operation (two or more bus

cycles) because of the store-merging mechanism. In this scenario, using a **dcbz** to initialize the line sometime before the stores occur may also improve performance. See [Section 3.1.2.3, “Store Gathering/Merging,”](#) for more information.

3.4.4.3 Data Cache Block Zero (dcbz)

On execution of a **dcbz** instruction, the effective address (EA) is computed, translated, and checked for protection violations as defined in the PowerPC architecture. The **dcbz** instruction is treated as a store to the addressed byte with respect to address translation, protection, and pipelining.

If the data is not found in the L1 or L2 caches as exclusive or exclusive-modified, the physical address is broadcast on the MPX bus prior to the zero line fill if $M = 1$. Note the following:

- If the address hits in the L1 as exclusive or exclusive modified, zeros are written to the cache and the tag is marked as exclusive modified.
- If the address hits in the L1 as shared or misses in the L1, a lookup is performed in the L2 cache.
- If the address hits in the L1 as shared and $M = 0$, the lookup in the L2 cache is ignored, zeros are written into the L1 cache, and the L1 tag is marked exclusive modified.
- If $M = 1$ and the L2 cache hits as exclusive or exclusive modified, zeros are written into the L1 and the L1 tag is marked exclusive modified.

Note that L1 cache misses for **dcbz** instructions follow the same line replacement algorithm as load misses to the L1 cache.

Executing a **dcbz** instruction can cause the following interrupts (noted in order of priority):

- Executing a **dcbz** instruction to a disabled or locked data cache generates an alignment interrupt.
- Executing a **dcbz** instruction to an EA with caching-inhibited or write-through attributes also generates an alignment interrupt.
- BAT and TLB protection violations for a **dcbz** instruction generate DSI interrupts.
- A **dcbz** instruction can also cause a data TLB miss on store interrupt if $HID0[STEN] = 1$ and either no translation is found in the BAT or TLB, or the changed bit in a matching TLB entry is cleared.

3.4.4.4 Data Cache Block Store (dcbst)

On execution of a **dcbst** instruction, the effective address is computed, translated, and checked for protection violations as defined in the PowerPC architecture. This instruction is treated as a load with respect to address translation and memory protection.

If the address hits in the cache and the cache block is in the modified state, the modified block is written back to memory and the cache block is placed in the invalid state in the L1. If the address hits in the data cache and the cache block is in any state other than modified, an address-only broadcast (**clean**) is performed and the cache block is placed in the invalid state in the data cache. If the address additionally hits in the L2 cache, the line is written back to memory and placed in the exclusive state in the L2.

The function of this instruction in the caches is independent of the WIMG bit settings of the block or PTE containing the effective address. However, if the address is marked memory-coherency- required ($M = 1$), the execution of **dcbst** causes an address broadcast on the MPX bus (if $HID1[ABE] = 1$). If

HID1[ABE] = 0, execution of **dcbst** only causes an address broadcast on the MPX bus if the data is modified. Execution of a **dcbst** instruction occurs whether or not the L1 or L2 caches are disabled or locked. However, it has no effect on a disabled L1 or L2 cache.

A BAT or TLB protection violation for a **dcbst** generates a DSI interrupt. Additionally, a **dcbst** instruction can also cause a data TLB miss on load interrupt if HID0[STEN] = 1 and no translation is found in the BAT or TLB.

3.4.4.5 Data Cache Block Flush (dcbf)

On execution of a **dcbf** instruction, the effective address is computed, translated, and checked for protection violations as defined in the PowerPC architecture. This instruction is treated as a load with respect to address translation and memory protection.

Note the following:

- If the address hits in the L1 or L2 cache, and the block is in the modified state, the modified block is written back to memory and the cache block is invalidated.
- If the address hits in the L1 or L2 cache, and the cache block is in the exclusive or shared state, the cache block is invalidated.
- If the address misses in the L1 or L2 cache, no action is taken.

The function of this instruction is independent of the WIMG bit settings of the block or PTE containing the effective address. However, if the address is marked memory-coherency- required, the execution of **dcbf** broadcasts an address-only FLUSH transaction on the MPX bus if HID1[ABE] = 1. Execution of a **dcbf** instruction occurs whether or not the L1 or L2 caches are disabled or locked. However, it has no effect on a disabled L1 or L2 cache.

A BAT or TLB protection violation for **dcbf** generates a DSI interrupt. Additionally, a **dcbf** instruction can also cause a data TLB miss on load interrupt if HID0[STEN] = 1 and no translation is found in the BAT or TLB. See [Section 3.5.8, “L1 Cache Invalidation and Flushing,”](#) for more information.

3.4.4.6 Data Cache Block Allocate (dcba)

The e600 core implements the Data Cache Block Allocate (**dcba**) instruction. This is currently an optional instruction in the PowerPC virtual environment architecture (VEA); however, it may become required in future versions of the architecture. The **dcba** instruction provides potential system performance improvement through the use of a software-initiated pre-store hit. This allows software to establish a block in the data cache in anticipation of a store into that block, without loading the block from memory.

The e600 core executes the **dcba** instruction the same as a **dcbz** instruction, with one interrupt. In cases when **dcbz** causes an interrupt, a **dcba** acts as a no-op unless the interrupt is a DSI for a data breakpoint match or to generate a software table search operation (with HID0[STEN] = 1). Note that the **dcba** instruction has no effect when the L1 cache is disabled or locked. If the effective address of a **dcba** instruction targets a direct-store segment (T = 1), the instruction will not no-op, and the results are undefined.

3.4.4.7 Data Cache Block Invalidate (dcbi)

When a **dcbi** instruction is executed, the effective address is computed, translated, and checked for protection violations as defined in the PowerPC architecture. This instruction is treated as a store with respect to address translation and memory protection.

This instruction is treated the same as a **dcbf** in the caches. The only difference between **dcbi** and **dcbf** on the e600 core is that the **dcbi** instruction is privileged.

A BAT or TLB protection violation for a **dcbi** translation generates a DSI interrupt.

3.4.4.8 Instruction Cache Block Invalidate (icbi)

The **icbi** instruction invalidates a matching entry in the instruction cache. During execution, the effective address for the instruction is translated through the data MMU and broadcast on the MPX bus using the memory-coherency attribute from translation if $HID1[ABE] = 1$. This instruction is treated as a load with respect to address translation and memory protection.

The e600 core always sends the **icbi** to the instruction cache for cache block address comparison and invalidation. The **icbi** instruction invalidates a matching cache entry regardless of whether the instruction cache is disabled or locked. The L2 cache is not affected by the **icbi** instruction.

An **icbi** instruction should always be followed by a **sync** and an **isync** instruction. This ensures that the effects of the **icbi** are seen by the instruction fetches following the **icbi** itself. For self-modifying code, the following sequence should be used to synchronize the instruction stream:

1. **dcbst** or **dcbf** (push new code from L1 data cache and L2 cache out to memory)
2. **sync** (wait for the **dcbst** or **dcbf** to complete)
3. **icbi** (invalidate the old instruction cache entry in this processor and, by broadcasting the **icbi** to the MPX bus, invalidate the entry in all snooping processors)
4. **sync** (wait for the **icbi** to complete its bus operation)
5. **isync** (re-sync this processor's instruction fetch)

The second **sync** instruction ensures completion of all prior **icbi** instructions. Note that the second **sync** instruction is not shown in Section 5.1.5.2, "Instruction Cache Instructions," in *The Programming Environments Manual*. This **sync** is required on the e600 core.

Since the **sync** instruction strongly serializes the core memory subsystem, performance of code containing several **icbi** instructions can be improved by batching the **icbi** instructions together such that only one **sync** instruction is used to synchronize all the **icbi** instructions in the batch.

3.5 L1 Cache Operation

This section describes the e600 core cache operations performed by the L1 instruction and data caches.

3.5.1 Cache Miss and Reload Operations

This section describes the actions taken by the L1 caches on misses for cacheable accesses. Also, it describes what happens on cache misses for cache-inhibited accesses as well as disabled and locked L1 cache conditions.

3.5.1.1 Data Cache Fills

The e600 core data cache blocks are filled (sometimes referred to as a cache reload) from the L2 cache or the core memory subsystem when cache misses occur for cacheable accesses, as described in [Section 3.1.2, “Load/Store Unit \(LSU\),”](#) and [Section 3.1.3, “Core Memory Subsystem Blocks.”](#)

When the data cache is disabled ($HID0[DCE] = 0$), the e600 core treats all data accesses as cache-inhibited (as if the memory coherency bit $I = 1$). Thus, even if the access would have hit in the cache, it proceeds to the core memory subsystem as cache-inhibited. When the data is returned, it is forwarded to the requesting execution unit, but it is not loaded into any of the caches.

From 0 to 8 ways of the data cache can be locked, as described in [Section 2.1.6.5.16, “Load/Store Control Register \(LDSTCR\),”](#) and all 8 ways can also be locked by setting $HID0[DLOCK]$. When at least one way is unlocked, misses are treated normally and they allocate in one of the unlocked ways on a reload. If all 8 ways are locked, load misses proceed to the core memory subsystem as normal cacheable accesses. In this case, the data is forwarded to the requesting execution unit when it returns, but it is not loaded into the data cache. If all 8 ways are locked, stores are sent to the core memory subsystem as cacheable but write-through (as if $W = 1$).

The accesses caused by the following instructions cause the e600 core to take a DSI interrupt when the data cache is disabled or completely locked:

- **lwarx** or **stwcx**.
- **dcbz**

Note that cache-inhibited stores do not access any of the caches. See [Section 3.5.3, “Store Miss Merging,”](#) for more information on the handling of cacheable store misses. Also, see [Section 3.6.4.1, “L2 Cache Miss and Reload Operations,”](#) for more information on L2 cache fills.

3.5.1.2 Instruction Cache Fills

The instruction cache provides a 128-bit interface to the instruction unit, so four instructions can be made available to the instruction unit in a single clock cycle on an L1 instruction cache hit. On a miss, the e600 core instruction cache blocks are loaded in one 32-byte beat from the L2 cache; the instruction cache is nonblocking, providing for hits under misses.

The instruction cache operates similarly to the data cache when all eight ways are locked. When the instruction cache is disabled ($HID0[ICE] = 0$), the instruction accesses bypass the instruction cache. However, unlike the data cache, these accesses are forwarded to the core memory subsystem as cacheable

and proceed to the L2 cache. When the instructions are returned, they are forwarded to the instruction unit but are not loaded into the instruction cache.

The instruction unit fetches a total of eight instructions at a time directly from the core memory subsystem for the following cases of cacheable instruction fetches:

- The instruction cache is disabled.
- The instruction cache is enabled, all 8 ways are locked, and the access misses in the L1 cache.

Note that the e600 core bursts out of reset.

The e600 core always uses burst transactions for instruction fetches. If the instruction cache is disabled ($HID0[ICE] = 0$), the core will perform a four-beat burst for instruction fetches and discard the last two beats. If the instruction cache is enabled ($HID0[ICE] = 1$), the core will perform a four-beat burst for instruction fetches and use all four beats. Externally, at the next I-fetch, the address will increment by 16 bytes if the instruction cache is disabled or the address will increment by 32 bytes if the instruction cache is enabled. For more details about disabling the instruction and data cache see [Section 2.1.6.1, “Hardware Implementation-Dependent Register 0 \(HID0\),”](#) and [Section 8.2.9, “Write-Through \(wt\), Cache Inhibit \(ci\), and Global \(gbl\) Signals.”](#)

Note that although the L1 instruction cache is physically addressed, the branch target instruction cache (BTIC) is virtually addressed. However, it is automatically flushed when the instruction cache is invalidated, when an interrupt occurs, or when a **tlbie**, **icbi**, **rfi**, or **isync** instruction is executed. Because the BTIC is automatically flushed any time the address mappings might change, aliases do not occur in the BTIC. See [Section 6.3.1, “General Instruction Flow,”](#) for more information on the BTIC.

3.5.2 Cache Allocation on Misses

This section describes the allocation of cache lines for both instruction and data cache misses. See [Section 3.5.7, “L1 Cache Block Replacement Selection,”](#) for more information on L1 cache block replacement. See [Section 3.6.4.2, “L2 Cache Allocation,”](#) for more information on the allocation and replacement algorithms used by the L2 cache.

3.5.2.1 Instruction Access Allocation in L1 Cache

Instruction cache misses cause a new line to be allocated into the instruction cache on a pseudo LRU basis, provided the cache is not completely locked or disabled.

3.5.2.2 Data Access Allocation in L1 Cache

Data load or write-back store accesses that miss in the L1 data cache function similarly to L1 instruction cache misses. They cause a new line to be allocated on a pseudo LRU basis, provided the cache is not completely locked or disabled.

Note that modified data in the replacement line of any of the caches can cause a castout to occur. In all of these cases, the castout is not initiated until the new data is ready to be loaded. Note that one data access can cause multiple castout operations to be initiated (from the various e600 core caches).

3.5.3 Store Miss Merging

Write-back stores that miss in the L1 data cache cause a data cache fill operation to occur using the load queues of the LSU. The store data is preserved internally, and when the remainder of the cache line has been loaded from the core memory subsystem, the store data is merged into the appropriate bytes of the cache line as it is loaded into the data cache. See [Section 3.1.2.3, “Store Gathering/Merging,”](#) for more information on store merging and [Section 3.6.4.3, “Store Data Merging and L2,”](#) for more information on store misses and the L2 cache.

3.5.4 Load/Store Miss Handling

The e600 core has support for two cacheable store misses, such that five loads/touches and/or two cacheable store misses (or **dcbz/dcba** fetches) can be in progress. Note that due to the constraint imposed by the size of the L1 castout queue (six entries), the maximum number of outstanding L1 data cache misses on the core remains six, but the ability to support an additional cacheable store miss prevents a stall from occurring if two successive stores miss in the L1. The e600 core can also have two instruction fetches (L1 instruction cache misses) and three L2 alternate sector hardware prefetches active in the core memory subsystem.

3.5.5 Store Hit to a Data Cache Block Marked Shared

When a write-back store hits in the L1 data cache and the block is shared, the target block is invalidated in the data cache. The current data from the target block is then treated as a store miss.

3.5.6 Data Cache Block Push Operation

When an L1 cache block in the e600 core is snooped (by another bus master) and the data hits and is modified, the cache block must be written to memory and made available to the snooping device. The push operation propagates out to the L2 cache, as well as the MPX bus. The cache block that hits is said to be pushed out onto the MPX bus.

3.5.7 L1 Cache Block Replacement Selection

Both the instruction and data cache use a pseudo least-recently-used (PLRU) replacement algorithm described in this section when a new block needs to be placed in the cache. Note that data cache replacement selection is performed at reload time, not when a miss occurs. Instruction cache replacement selection occurs when an instruction cache miss is first recognized.

3.5.7.1 PLRU Replacement

Each L1 cache is organized as eight blocks (ways) per set by 128 sets. There is an identifying bit for each way in the cache, L[0–7]. The PLRU algorithm is used to select the replacement target. There are seven PLRU bits, B[0–6] for each set in the cache.

This algorithm does not prioritize replacing invalid entries over valid ones; a way is selected for replacement according to the PLRU bit encodings shown in [Table 3-7](#).

Table 3-7. L1 PLRU Replacement Way Selection

If the PLRU bits are:					Then the way selected for replacement is:	
B0	0	B1	0	B3	0	L0
	0		0	B3	1	L1
	0		1	B4	0	L2
	0		1		B4	1
	1	B2	0	B5	0	L4
	1		0		B5	1
	1		1	B6	0	L6
	1		1		B6	1

The PLRU algorithm is shown graphically in [Figure 3-16](#).

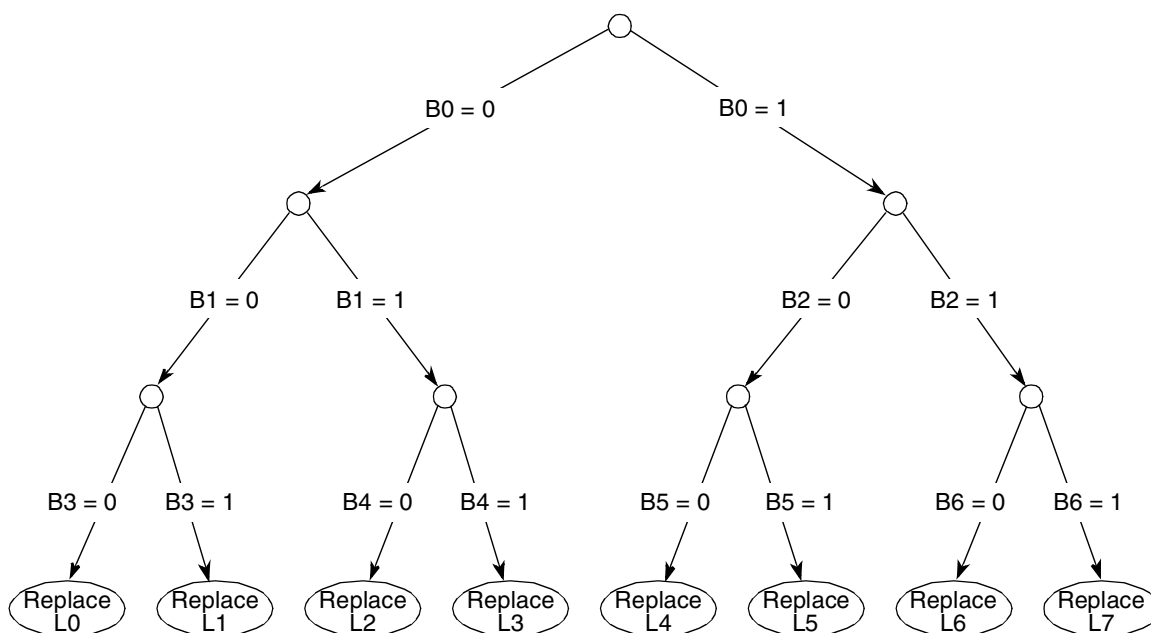


Figure 3-16. PLRU Replacement Algorithm

During power-up or hard reset, the valid bits of the L1 caches are not necessarily cleared and they must be explicitly cleared by setting the respective flash invalidate bits (HID0[DCFI] or HID0[ICFI]) before each cache is enabled. Subsequently, the PLRU bits are cleared to point to way L0 of each set.

3.5.7.2 PLRU Bit Updates

Except for snoop accesses, each time a cache block is accessed, it is tagged as the most recently used way of the set (unless accessed by the AltiVec LRU instructions; refer to [Section 7.1.2.1, “LRU Instructions”](#)). For every hit in the cache or when a new block is reloaded, the PLRU bits for the set are updated using the rules specified in [Table 3-8](#). Note that only three PLRU bits are updated for any given access.

Table 3-8. PLRU Bit Update Rules

If the current access is to:	Then the PLRU bits in the set are changed to the following ¹ :						
	B0	B1	B2	B3	B4	B5	B6
L0	1	1	x	1	x	x	x
L1	1	1	x	0	x	x	x
L2	1	0	x	x	1	x	x
L3	1	0	x	x	0	x	x
L4	0	x	1	x	x	1	x
L5	0	x	1	x	x	0	x
L6	0	x	0	x	x	x	1
L7	0	x	0	x	x	x	0

¹ x = Does not change.

3.5.7.3 AltiVec LRU Instruction Support

The data cache fully supports the AltiVec LRU instructions (**lvxl**, **stvxl**). If one of these instructions causes a hit in the data cache, the PLRU bits are updated such that the way which hit is marked as least-recently-used by using the PLRU update rules shown in [Table 3-9](#). If no other hit to the cache index occurs, this way is selected for replacement upon the next data cache reload. Similarly, if an **lvxl** or **stvxl** instruction misses in the cache, the PLRU bits are updated, as shown in [Table 3-9](#), when that cache block reloads the data cache. Note that the instruction cache is not subject to any AltiVec LRU accesses.

Table 3-9. PLRU Bit Update Rules for AltiVec LRU Instructions

If the current AltiVec LRU access is to:	Then the PLRU bits in the set are changed to the following ¹ :						
	B0	B1	B2	B3	B4	B5	B6
L0	0	0	x	0	x	x	x
L1	0	0	x	1	x	x	x
L2	0	1	x	x	0	x	x
L3	0	1	x	x	1	x	x
L4	1	x	0	x	x	0	x
L5	1	x	0	x	x	1	x
L6	1	x	1	x	x	x	0
L7	1	x	1	x	x	x	1

¹ x = Does not change.

Note that an AltiVec LRU access simply inverts the update value of the three PLRU bits when compared to the normal (most-recently-used) update rules.

3.5.7.4 Cache Locking and PLRU

Care should be taken when locking between 1 and 8 ways in either of the L1 caches. For the best performance, there should be an equal number of locked ways on each side of each decision point of the binary tree shown in [Figure 3-16](#), or all ways should be locked. Otherwise, the PLRU replacement algorithm will be biased to replace certain ways.

3.5.8 L1 Cache Invalidation and Flushing

When software guarantees that memory is not shared, the data cache can be invalidated by executing a series of loads followed by **dcbf** (or **dcbi**) instructions or by setting `HID0[DCFI]`. The instruction cache can be invalidated by setting `HID0[ICFI]`.

When coherency is required to be maintained and data is shared among caches in a system, and the cache is going to be disabled or reconfigured, all the modified data in the data cache can be flushed by executing the following instructions in this order:

1. Way n:
 - a. Start with a base offset of zero. Perform a load followed by a **dcbf** instruction to that same address.
 - b. Increment the base offset by 32 bytes and perform the load/**dcbf** pair to the new address.
 - c. Repeat step b 126 more times so that each load/**dcbf** pair addresses a different cache line in a way (progressing through all 128 combinations of `PA[24:30]`), assuming 36-bit physical addressing).
2. Way n + 1: Repeat the process shown in step 1 for the next way in the cache. This is started by incrementing the base offset used for the last set in way n by 32 bytes. Now `PA[20:23]` is incremented by one. Then repeat the remainder of step 1.
3. Way n + 2 to way n + 7: Repeat the process described in step 2 six more times (effectively progressing through all 8 combinations of `PA[20:23]`).

The **dcbf** instructions described above are not required if the loads in the sequence can be guaranteed to replace (flush) all the modified data in the cache and the loads can be from known addresses that will not be modified. This can be accomplished by loading from a memory range that will not be modified.

Interrupts and other events that can cause an L1 cache access should be disabled during this time so that the PLRU algorithm can function undisturbed. However, if it is impossible to disable interrupts and other events that can affect the PLRU, the sequence shown above can be modified as follows:

- Lock all ways in the data cache except way n. Then perform the process in step 1 above.
- Lock all ways in the data cache except way n + 1 and perform step 2, continuing with step 3 by unlocking way n + 2 through way n + 7 and performing the load/**dcbf** pairs for each unlocked way, one way at a time.

To minimize the time required to flush all the caches in the e600 core, the L1 data cache can be flushed before flushing the L2 cache, thus eliminating the flushing of the same line multiple times if it is modified in both the L1 and the L2 caches. Note that if cache flushing is performed without using the **dcbf** instruction and the L2 is flushed before the L1, the L2 should be disabled before flushing the L1 cache.

This avoids loading of modified data into the L2. See 3.6.3.1.5, “Flushing of L1 and L2 Caches,” for more information on flushing the L2 cache.

3.5.9 L1 Cache Operation Summary

Table 3-11 summarizes all L1 cache activities caused by internal conditions. Table 3-10 defines some of the abbreviations used in Table 3-11. Note that the WIMG bits are passed onto the core memory subsystem unless explicitly shown as overridden in the MSS request type column of Table 3-11. See Section 3.7.3.2, “L1 Cache State Transitions and Bus Operations Due to Snoops,” for a detailed description of L1 cache state transitions caused by MPX bus snooping.

Table 3-10. Definitions for L1 Cache-State Summary

Term	Definition
Load	One of the following instructions: lbz , lbzx , lbzu , lbzux , lhz , lhzx , lhzu , lhzux , lha , lhax , lhau , lhaux , lwz , lwzx , lwzu , lwzux , lhrx , lwbrx , lmw , lswi , lswx , lvebx , lvehx , lvewx , lvx , lvxl , lvsl , and lvslr . A load reads cache memory and returns a data value of between 1 and 16 bytes. If the data is not in the L1 cache, the access causes a request to lower cache/memory to reload the L1 cache with the 32-byte cache line containing the requested data. If the 8/16 bytes of data (depending on size) containing the requested data are available before the rest of the cache line, this critical double-word is forwarded to the requesting execution unit before the line is reloaded. Note that misaligned loads and load string or load multiple may cause multiple memory accesses.
Store	One of the following instructions: stb , stbx , stbu , stbux , sth , sthx , sthu , sthux , stw , stwx , stwu , stwux , sthrx , stwbrx , stmw , stswi , stswx , stvebx , stvehx , stvewx , stvx , stvxl . Stores cause an update of cache and/or memory of 1–16 bytes of data, depending on the WIMG settings. Stores may cause a reload similar to loads above. Stores do not cause forwarding of data. Note that misaligned stores and store string or store multiple may cause multiple memory accesses.
Touch	One of the following instructions: dcbt or dst . Touches may cause a reload similar to loads above. Touches do not cause forwarding of data. Note that data stream touch (dst) may cause multiple memory accesses.
Store Touch	One of the following instructions: dcbtst or dstst . Store touches may cause a reload similar to loads above. Store touches do not cause forwarding of data. Note that data stream touch for store (dstst) may cause multiple memory accesses
dss	Data stream stop. It causes the tagged stream to stop prefetching. It is not sent to the MSS, and has no effect on prefetch requests already sent to the MSS.
lwarx	The same as loads above, but also causes the setting of the reservation bit in the processor.
stwcx.	The same as stores above, but the store is not performed unless the reservation is set, and the reservation is cleared once the store passes the coherency point.
dcbst , dcbf	Push modified data from any processor out to memory, and change valid lines to invalid.
dcbz , dcba	Claims ownership of a line without reloading the data and zeroes out the line.
L1 Deallocate	Caused by the allocation of a line in the L1 for a reload or dcbz . A deallocation casts out modified data and invalidates the line.
MSS request and MSS response	Core memory subsystem request and core memory subsystem response
same	The state is unchanged.
x	Do not care.
n/a	Does not apply.

Table 3-11. L1 Cache-State Transitions and MSS Requests

Internal Operation	WIMG Setting	Initial L1 State	MSS Request	MSS Response	Final L1 State	Comments
Load	I = 0	I	Load	S	S	Load miss. Deallocate a line in the cache and reload the missing one from the MSS.
				E	E	
		S/E/M	None	N/A	Same	Load hit—return data from L1.
	I = 1	N/A	Load	N/A	N/A	Cache-inhibited load
dcbt/dst	I = 0	I	Touch	S	S	Touch miss. Deallocate a line in the cache and reload the missing one from the MSS
				E	E	
		S/E/M	None	N/A	Same	No-op
	I = 1	N/A	None	N/A	N/A	No-op
dcbtst/dsts	I = 0	I/S	Store Touch	S	S	Store touch miss. Reload the missing/shared one from the MSS if missing; deallocate a line.
				E	E	
		E/M	None	N/A	Same	No-op
	I = 1	N/A	None	N/A	N/A	No-op
dss	x	N/A	None	N/A	N/A	Stops a dst or dsts .
lwarx	I = 0	I	LWARX	S	S	Same as load, but atomic bit is set on MSS access and reservation is set.
				E	E	
		S/E/M	None	N/A	Same	Same as load hit, but set reservation
	I = 1	N/A	None	N/A	N/A	Cache-inhibited lwarx causes DSI interrupt.
	W = 1	N/A	None	N/A	N/A	Write-through lwarx causes DSI interrupt.
Store	W = 0 I = 0	I	Store	N/A	M	If L1 = I, deallocate a cache line and reload data from MSS. If L1 = S, invalidate and allocate line before initiating RWITM. Store data is merged with reload data.
	M = X W = 0 I = 0	S	Store	N/A	M	
	W = 0 I = 0	E/M	None	N/A	M	Merge store data into L1.
	I = 1	n/a	Store	N/A	N/A	Initiate a store request to the MSS without changing cache state.
	W = 1	I/S/E/M	Store	N/A	N/A	If line is valid, merge store data into L1. Initiate a store request to MSS.

Table 3-11. L1 Cache-State Transitions and MSS Requests (continued)

Internal Operation	WIMG Setting	Initial L1 State	MSS Request	MSS Response	Final L1 State	Comments	
stwcx.	W = 0 I = 0	I	STWCX	E	M	Same as stores, but do not store data if reservation is not set. Reset reservation when past coherency point. Return whether successful. Note: a stwcx. which loses its reservation while pending in the MSS is converted into a load and possibly returns a shared response.	
				S	S		
	M = 0 W = 0 I = 0	S	None	N/A	M		
	W = 0 I = 0	E/M	None	N/A	M		
W = 1	N/A	None	N/A	N/A	Write-through stwcx. causes DSI interrupt.		
					dcbst	x	
M	Write w/Clean	N/A	I				
dcbf	x	I/S/E	DCBF	N/A	I	Push any modified data out to memory and leave cache line invalid.	
		M	Castout (W = 1)	N/A	I		
dcbz	I = 0 W = 0	I	DCBZ	N/A	M	Zero out data	
							M = X I = 0 W = 0
	I = 0 W = 0	E/M	None	N/A	M	Zero out data.	
							W = 1
	I = 1	N/A	None	N/A	N/A		
dcba						I = 0 W = 0	N/A
	M = X I = 0 W = 0	S	DCBZ	N/A	M		
						I = 0 W = 0	E/M
	W = 1	N/A	None	N/A	N/A		
						I = 1	N/A

Table 3-11. L1 Cache-State Transitions and MSS Requests (continued)

Internal Operation	WIMG Setting	Initial L1 State	MSS Request	MSS Response	Final L1 State	Comments
L1 Deallocate	x	I/S/E	None	N/A	I	Deallocate is caused by an operation to another address (e.g. load) requiring an allocation of a cache line. Cast out modified data and invalidate line.
	x	M	Castout (W = 0)	N/A	I	
icbi	x	N/A	ICBI	N/A	N/A	No effect on D cache.
tlbie	x	N/A	TLBIE	N/A	N/A	No effect on L1 caches.
tlbsync	x	N/A	TLBSYNC	N/A	N/A	
sync	x	N/A	SYNC	N/A	N/A	
eieio	x	N/A	EIEIO	N/A	N/A	

3.6 L2 Cache

This section provides information about the L2 cache on the e600 core. It describes the L2 cache organization, L2 features and how they are controlled, L2 cache operation, and provides a summary of all actions of the L2 caused by internal operations in a summary table. See [Section 3.7.3.3, “L2 Operations Caused by Snoops External to the Core,”](#) for more information about the L2 cache and bus snooping.

3.6.1 L2 Cache Organization

The integrated L2 cache is organized as shown in [Figure 3-17](#).

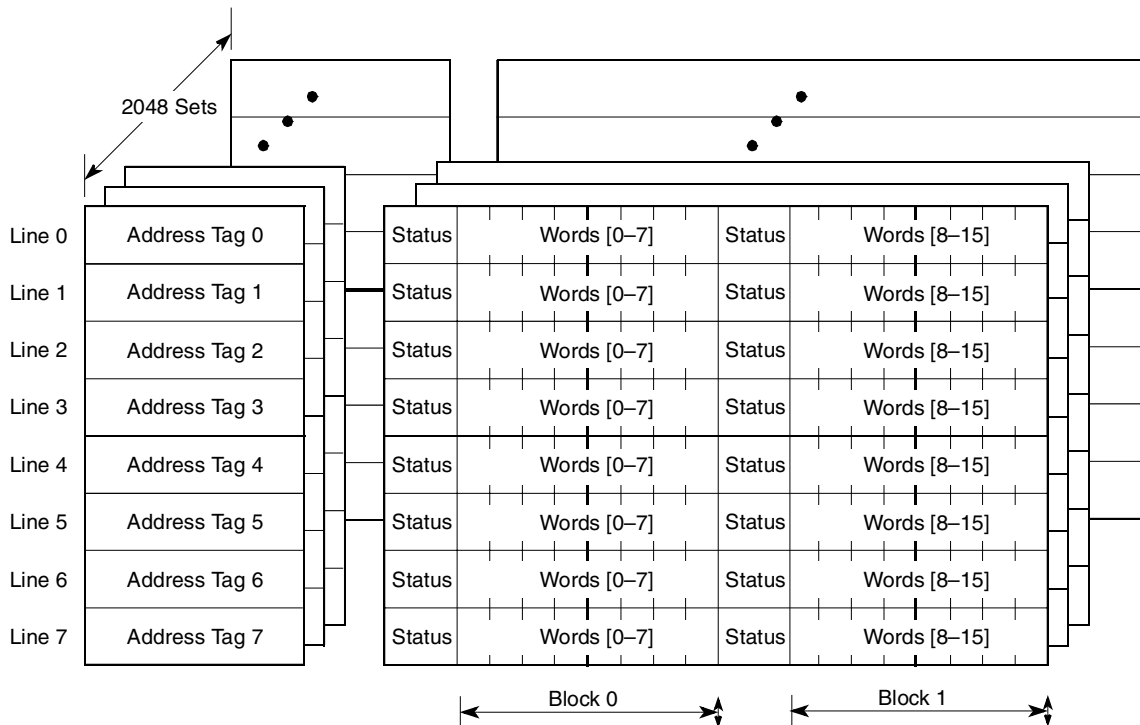


Figure 3-17. L2 Cache Organization

Each line consists of 64 bytes of data, organized as two blocks (also called sectors) that are selected by one address bit. Although all 16 words in a cache line share the same address tag, each block maintains the two separate status bits for the 8 words of the cache block, the unit of memory at which coherency is maintained. Thus, each cache line can contain 16 contiguous words from memory that are read or written as 8-word operations. Note that the line replacement information for the L2 cache is maintained on a line basis.

The L2 cache tags are fully pipelined and non-blocking for efficient operation. Thus, the L2 cache can be accessed internally while a load for a miss is pending (allowing hits under misses). A reload for a cache miss is treated as a normal access and blocks other accesses for only a single cycle.

Similar to the L1 data cache, there are two status bits associated with each cache block of the L2 cache. These bits are used to implement the modified/exclusive/shared/invalid (MESI) cache coherency protocol. The coherency protocols are described in [Section 3.3, “Memory and Cache Coherency.”](#)

3.6.2 L2 Cache and Memory Coherency

The e600 core models for memory and cache coherency described in [Section 3.3, “Memory and Cache Coherency,”](#) for the L1 caches all apply for the L2 cache. Specifically, the WIMG bit model, the MESI cache coherency protocol, and the architectural implications of the ordering of loads and stores are as described in that section.

3.6.3 L2 Cache Control

The parameters for the L2 cache are controlled by L2CR, MSSCR0, and MSSSR0.

3.6.3.1 L2CR Parameters

The L2CR enables the L2 cache, enables parity checking on the L2, provides for instruction-only and data-only modes, provides hardware flushing for the L2, and selects between two available replacement algorithms for the L2 cache. L2CR is a supervisor-level read/write, implementation-specific register that is accessed as SPR 1017. The contents of L2CR are cleared during power-on reset. Refer to [Section 2.1.6.5.1, “L2 Cache Control Register \(L2CR\),”](#) for the bit descriptions of L2CR.

3.6.3.1.1 Enabling the L2 Cache and L2 Initialization

When the L2 cache is disabled, all accesses bypass the L2. Before the L2 cache is enabled, all L2 cache configurations must be set appropriately in L2CR and the L2 tags must be invalidated in the following sequence:

1. Verify that $L2CR[L2E] = 0$.
2. Invalidate the entire L2 cache by setting $L2CR[L2I]$. See [Section 3.6.3.1.4, “L2 Cache Invalidation.”](#)
3. Poll $L2CR[L2I]$ until it is cleared.
4. Set remaining desired bits in L2CR and then set $L2CR[L2E]$.

The L2 cache is disabled out of reset, so $L2CR[L2E] = 0$. Note that out of reset, the sequence above must obviously be preceded by the assertion and negation of *hreset*.

Setting L2CR[L2E] enables operation of the L2 cache, including snooping of the L2. Note that the **dcbf**, **dcbst**, and **dcbi** instructions have no effect on the L2 cache when it is disabled.

3.6.3.1.2 Enabling L2 Parity Checking

The L2 cache maintains one parity bit per byte of data and an additional parity bit for each tag (one tag parity bit per line).

In the e600 core, L2 cache parity checking is enabled in two separate registers, L2CR and L2ERRDIS. L2 cache data parity is enabled by setting L2CR[L2PE], and tag parity is enabled by clearing L2ERRDIS[TPARDIS]. When L2CR[L2PE] = 1, L2 data parity bits are generated and checked; when L2ERRDIS[TPARDIS] = 0, L2 tag parity bits are generated and checked. ECC must be disabled for data parity checking to be enabled with L2CR[L2PE].

When an L2 data or tag parity error occurs for either the L2 address or data buses, a machine check interrupt is generated if MSR[ME] = 1. Otherwise, if MSR[ME] = 0, a checkstop occurs. Note that as a result of a machine check interrupt caused by an L2 parity error, SRR1[11] is set, and enabled L2 data parity errors are reported in the L2TAG and L2DAT bits of MSSSR0. See [Section 3.6.3.3, “L2 Parity Error Reporting,”](#) and [Section 2.1.6.4, “Memory Subsystem Status Register \(MSSSR0\),”](#) for detailed information on the MSSSR0 bits. Enabled L2 tag parity errors are reported in the TPARERR bit of the error detect register (L2ERRDET) as well as in MSSSR0. See [Section 3.6.3.4, “L2 Data ECC,”](#) for information on ECC support in the e600 core.

NOTE

If ECC is enabled, setting L2CR[L2PE] has no effect; ECC checking will continue.

3.6.3.1.3 L2 Instruction-Only and Data-Only Modes

The L2CR also maintains the L2IO and L2DO bits for limiting the types of new accesses that are allocated into the L2. When L2CR[L2IO] is set, only instruction accesses that miss in the L2 allocate new entries in the L2. Data accesses that hit (loads and stores) operate normally (except for the case of store hits to blocks marked shared that actually function as misses). When L2CR[L2DO] is set, only data accesses that miss in the L2 allocate new entries in the L2. Instruction accesses that are already resident in the L2 (allocated before L2DO was set) provide instructions normally.

If both L2IO and L2DO are set, the L2 is effectively locked, and no new entries are allocated.

3.6.3.1.4 L2 Cache Invalidation

The L2 cache can be globally invalidated by setting L2CR[L2I]. This causes all valid bits in the L2 cache to be cleared. The L2CR[L2I] bit must not be set while the L2 cache is enabled (L2CR[LE] = 0). When the e600 core completes the invalidation, L2CR[L2I] is automatically cleared.

When software sets L2CR[L2I], the L2 cycles through all the tags and invalidates every entry in the cache without regard to the state of the line. The processor clears L2CR[L2I] upon completing the invalidation of the entire cache. Software can poll L2CR[L2I] to know when the invalidation is complete.

The sequence for performing a global invalidation of the L2 cache is as follows:

1. Execute a **dssall** instruction to cancel any pending data stream touch instructions.
2. Execute a **sync** instruction to finish any pending store operations in the load/store unit, disable the L2 cache by clearing L2CR[L2E], and execute an additional **sync** instruction after disabling the L2 cache to ensure that any pending operations in the L2 cache unit have completed.
3. Initiate the global invalidation operation by setting the L2CR[L2I] bit.
4. Monitor the L2CR[L2I] bit to determine when the global invalidation operation is completed (indicated by the clearing of L2CR[L2I]). The global invalidation requires approximately 8K core clock cycles to complete.
5. After detecting the clearing of L2CR[L2I], re-enable the L2 cache for normal operation by setting L2CR[L2E].

3.6.3.1.5 Flushing of L1 and L2 Caches

The e600 core provides a hardware flushing mechanism for the L2 through the L2CR[L2HWF] bit. Note that prior to flushing the caches, L2 prefetching must be disabled (MSSCR0[L2PFE] = 0). When L2CR[L2HWF] is set, the L2 begins a flush by starting with the first cache index. Each modified block (sector) is cast out as it is flushed. After the first line in the first way is flushed (one block and then the other), the next way (same index) is flushed. When all ways for a given index have been flushed, the index is incremented and same process occurs for line 1, and so on.

During a hardware flush, the L2 services both read hits and bus snooping.

The hardware flush completes when all blocks in the L2 have a status of invalid. At this time, the processor automatically clears L2CR[L2HWF]. However, even though the hardware flush is considered complete, there may still be outstanding castouts in the BSQ waiting to be performed to the core interface.

Note that to guarantee that the L2 is completely invalid when flushing is complete, software must ensure that the L2 does not allocate new entries while the L2 is being flushed, by setting both L2CR[L2IO] and L2CR[L2DO] to lock the L2 cache.

The L2CR[L2I] invalidation is a subset of the L2CR[L2HWF] flushing mechanism. The following sequence of steps is recommended for flushing the L1 and L2 caches in the e600 core:

1. Disable external interrupts (clear MSR[EE] to guarantee that the PLRU for the L1 is undisturbed by an interrupt handler).
2. Disable L2 prefetching (clear MSSCR0[L2PFE]).
3. Flush the L1 data cache as described in [Section 3.5.8, “L1 Cache Invalidation and Flushing.”](#)
4. Set the L2CR[L2IO] and L2CR[L2DO] bits to completely lock the L2 cache.
5. Perform an **mtspr** L2CR to set L2HWF.
6. Poll the L2CR[L2HWF] bit using **mfspir** L2CR until L2CR[L2HWF] is cleared. When the bit is cleared, issue a **sync**.
7. Clear the L2CR[L2IO] and L2CR[L2DO] bits.

Also note that because the e600 core shares the invalidation and flushing logic internally, it is a programming error to set both L2CR[L2I] and L2CR[L2HWF] at the same time. Setting more than one of these bits at any one time can cause the L2 cache to not fully invalidate.

3.6.3.1.6 L2 Replacement Algorithm Selection

The L2 cache supports two pseudo-random modes of line replacement, selected by L2CR[L2REP]—3-bit counter mode, and pseudo-random number generator mode. See [Section 3.6.4.4, “L2 Cache Line Replacement Algorithms,”](#) for a detailed description of the two L2 replacement algorithms.

3.6.3.2 L2 Prefetch Engines and MSSCR0

Depending on the application, it may enhance performance to prefetch the second block of an L2 cache line on a cache line miss, even if no data in the second block is currently required. In this case, from one to three prefetch engines can be enabled to fill invalid blocks (that share a line with a valid block) in the L2 cache.

The L2 prefetch engines are enabled through the MSSCR0[L2PFE] field. Note that it is an error to enable the prefetch engines when the L2 cache is disabled. When L2 prefetching is enabled, a prefetch is initiated when a load, instruction fetch, or write-back store misses in all the caches and the transaction must be performed to the external core interface for the required block. In this case, a prefetch is initiated to fill the second (unrequired) block, provided an enabled prefetch engine is available.

However, L2 prefetches are not initiated if:

- The access is a data cache miss and the L2 cache is set up to cache instructions only (L2CR[L2IO] = 1) or
- The access is an instruction cache miss and the L2 cache is set up to cache data only (L2CR[L2DO] = 1).

Note that prior to flushing the caches MSSCR0[L2PFE] must be cleared; see [Section 3.6.3.1.5, “Flushing of L1 and L2 Caches,”](#) for further details on how to flush the caches.

3.6.3.3 L2 Parity Error Reporting

When L2 cache parity checking is enabled, L2 tag and data parity bits are independently generated and checked. Enabled L2 tag and data parity errors are reported in the MSSSR0[L2TAG] and MSSSR0[L2DAT] register fields. In addition to MSSSR0[L2TAG], the enabled L2 tag parity errors are also reported in L2ERRDET[TPARERR]. See [Section 3.6.3.1.2, “Enabling L2 Parity Checking,”](#) for more information.

3.6.3.4 L2 Data ECC

In the e600 core, L2 error detection, reporting, and injection allow flexible handling of ECC and parity errors in the L2 data and tag arrays. When the core detects an L2 error, the appropriate bit in the error detect register (L2ERRDET) is set. Error detection is disabled by setting the corresponding bit in the error disable register (L2ERRDIS). By default, tag parity and data ECC checking are enabled on the e600 core.

However, for a detected error to signal a 0x200 machine check, the user must enable error reporting in error interrupt enable register (L2ERRINTEN) for each error.

Whereas setting L2CR[L2PE] in the enables data parity checking, tag parity in the e600 core is controlled separately through the L2ERRDIS[TPARDIS] bit. Data parity in the e600 core can only be enabled through L2CR[L2PE] if ECC is disabled in the L2ERRDIS register.

3.6.3.4.1 Enabling or Disabling ECC

The L2 cache must be disabled and flushed before enabling or disabling ECC to ensure that no errors occur. ECC can be enabled or disabled with the L2ERRDIS register. Refer to [Section 2.1.6.5.9, “L2 Error Disable Register \(L2ERRDIS\),”](#) for details on enabling or disabling ECC with the L2ERRDIS register. The proper sequence for enabling or disabling ECC is as follows:

```
sync
mtspr
sync
isync
```

3.6.3.4.2 L2 Error Control and Capture

The error control and capture registers control the detection and reporting of tag parity and ECC errors. Error reporting (by generating an interrupt) is enabled by setting the corresponding bit in the error interrupt enable register (L2ERRINTEN). Note that the error detect bit is set regardless of the state of the interrupt enable bit.

Single-bit errors are corrected as data is read from the L2 cache and the corrected data is forwarded to the appropriate destination. Note that the copy of the data stored in the L2 cache is not corrected. Because single-bit errors may accumulate in the L2 over time and become multiple-bit errors, software may choose to periodically flush the L2 (single-bit errors are corrected as they are flushed from the L2) or take other measures to correct data stored in the L2 once a certain number of single-bit errors are detected. The L2 error control register (L2ERRCTL) maintains the L2 cache threshold for the number of ECC single-bit errors to be detected before reporting an error condition and a count of the number of ECC single-bit errors that have already been detected. If the count reaches the threshold and single-bit error reporting is enabled (SBECCDIS = 0 and SBECCINTEN = 1), an error is reported.

Double-bit errors are detected but not corrected. In most instances, errors of three or more bits are detected; however, it is not guaranteed that multiple-bit errors will always be detected. For all multiple-bit errors, including double-bit errors, note that the erroneous data is still forwarded to its destination, and it is software's responsibility to take appropriate corrective action if an error is reported. The double word where an error resides is always known because error detection is performed on a double-word boundary.

The data, address, and attributes of the first detected error that causes an error to be reported are saved in the error capture registers (L2ERRADDR, L2ERRATTR, L2CAPTDATAHI, L2CAPTDATALO, and L2CAPTECC). These registers are updated every time an error is detected, but are frozen after the detection of an error that causes an error to be reported (L2ERRATTR[VALINFO] = 1). Subsequent errors set error bits in the error detection registers, but only the first error that was reported (by generating an interrupt) has the associated information captured, until software unlocks error capture by clearing L2ERRATTR[VALINFO] to 0. L2CAPTDATAHI and L2CAPTDATALO hold the high and low words of the L2 data that contains the detected error. The calculated ECC syndrome and datapath ECC of the failing

double word are saved in the L2 error syndrome register (L2CAPTECC). The type of error detected (multiple L2, tag parity, multiple-bit ECC, single-bit ECC) is reflected in L2ERRDET. Tag parity, multiple-bit, and single-bit error detection can be enabled or disabled in L2ERRDIS.

L2ERRDET is implemented as a bit-reset type register. Reading from this register occurs normally; however, write operations can clear but not set bits. A bit is cleared whenever the register is written and the data in the corresponding bit location is a 1. For example, to clear bit 6 and not affect any other bits in the register, the value 0x0200_0000 is written to the L2ERRDET register.

3.6.3.4.3 ECC Error Reporting

When ECC is enabled and an ECC error is detected, the error is reported in both MSSSR0 and L2ERRDET. The ECC error is reported in the MSSSR0[L2DAT] field, as well as either L2ERRDET[SBECCERR] or L2ERRDET[MBECCERR], depending on whether the error is a single-bit or multiple-bit error.

3.6.3.4.4 L2 Error Injection

The L2 cache includes support for injecting errors into the L2 data, data ECC or tag. This may be used to test error recovery software by deterministically creating error scenarios.

The preferred method for using error injection is to map all blocks or pages cache-inhibited (WIMG = x1xx), except for one that will be marked cacheable (WIMG = x0xx) to be used as a temporary scratch buffer, set L2CTL[L2DO] to prevent allocation of instruction accesses, and invalidate the L2 by setting L2CTL[L2I] = 1. The following code sequence triggers an error, and then detects it (A is an address in the scratch page):

```
lock 7 of 8 ways through LDSTCR[DCWL]
dcbz A
store pattern to A
dcbz B      | addr A + 0x1000 to hit in same L1 data cache set/way, and push addr A
            | to L2, injecting errors
lwz A      | reload from L2 to L1 data cache, causing injected errors to be detected
```

Data or tag errors are injected into the line, per the error injection settings in L2ERRINJHI, L2ERRINJLO, and L2ERRINJCTL, at allocation. The final load detects and reports the error (if enabled) and allows software to examine the offending data, address, and attributes.

Note that error injection enable bits in L2ERRINJCTL must be cleared by software and the L2 must be invalidated (by setting L2CTL[L2I]) before resuming L2 normal operation.

3.6.3.5 Instruction Interactions with L2

The following instructions have effects on the L2 cache as listed:

- **dcbz** and **dcba** instructions that miss or hit as shared cause L2 allocation to reserve the line and a kill is sent to the MPX bus. When the kill completes, the L2 line is marked exclusive. **dcbz** instructions that hit as modified or exclusive cause no L2 state change.

- On the e600 core, **dcba** differs from **dcbz** only in its interrupt generation. As such, it is identical to **dcbz** from an L2 perspective.
- Line pushes from the L1 data cache as the result of **dcbf/dcbst** instructions write through to the MPX bus. **dcbf** invalidates the L2 cache block in case of hit. A **dcbst** hit does not affect the block if it hits as either shared or exclusive; it is changed to exclusive if it hits as modified.
- **dcbf/dcbst** instructions that do not require a castout from the L1 data cache are issued to the L2 cache and perform an invalidate and/or castout from the L2 cache to the MPX bus as required. If they do not require a castout from the L2 cache, they are also issued to the MPX bus.
- **dcbf** and **dcbi** instructions that address an area of memory marked with $M = 1$ cause a global transaction on the MPX bus if the line is modified or if $HID1[ABE]$ is set.
- **icbi** instructions bypass the L2 cache and are forwarded to the MPX bus.
- **sync** and **eiio** instructions bypass the L2 cache, and are forwarded to the MPX bus for further processing. Also, all **sync** and **eiio** instructions are broadcast on the MPX bus if $HID1[SYNCBE] = 1$.
- **tlbie** and **tlbsync** instructions bypass the L2 cache, and are forwarded to the core interface for further processing.
- **dcbf**, **dcbst**, **dcbi**, **icbi**, **tlbie**, and **tlbsync** instructions are broadcast on the MPX bus if $HID1[ABE] = 1$.

3.6.4 L2 Cache Operation

This section describes the e600 core L2 cache operations.

All accesses to the L2 cache that are marked cache-inhibited bypass the L2 cache (even if they would have normally hit), and do not cause any L2 state changes. Note that all data accesses performed while the L1 data cache is disabled are considered cache-inhibited by the L2 cache and the rest of the core memory subsystem. Therefore, all read accesses from the L2 cache are caching-allowed and burst accesses (32-byte reads).

Single-beat writes occur to the L2 cache for the following:

- Write-through ($W = 1$) accesses that hit in the L2
- Stores that hit in the L2 if all ways of the L1 cache are locked with $LDSTCR[DCWL]$
- Stores that hit in the L2 if the L1 data cache is completely locked with $HID0[DLOCK] = 1$

Accesses that meet one of these three conditions always propagate to the MPX bus as write-through stores.

For write-through stores ≥ 8 bytes that hit in the L2, the line is updated in the L2, but the status bits for the block are not changed, and the write-through store propagates to the MPX bus. Otherwise, for write-through stores < 8 bytes, if the L2 block is modified, the block is cast out to the MPX bus and the write-through store propagates to the MPX bus; if the L2 block is not modified, the block is invalidated, and the write-through store propagates to the MPX bus.

In case of multiple pending requests to the L2 cache, the priorities are as shown in [Table 3-12](#).

Table 3-12. L2 Cache Access Priorities

Priority	Type of Access
1	Snoop request
2	Reload into L2 or L1
3	L2 castout
4	Snoop push or data intervention
5	In the following order: a. Cacheable store miss in the L1 data cache b. Load miss in the L1 data cache c. Instruction miss in the L1 instruction cache
6	L1 castout

This section contains a detailed description of L2 actions caused by L1 requests. For more information on L2 actions caused by bus snooping, see [Section 3.7.3.3, “L2 Operations Caused by Snoops External to the Core.”](#)

3.6.4.1 L2 Cache Miss and Reload Operations

The e600 core L2 cache blocks are filled (sometimes referred to as a cache reload) from the core memory subsystem when cache misses occur for cacheable accesses, as described in [Section 3.1.2, “Load/Store Unit \(LSU\),”](#) and [Section 3.1.3, “Core Memory Subsystem Blocks.”](#)

As an L2 cache line is received from the MPX bus it is loaded into the L2 cache and marked according to the snoop response. If the reload requires a new line to be allocated in the L2 cache and the current line is modified, the modified line is castout from the L2 cache to the MPX bus at the time of the miss (not at the time of the reload).

Note that the L2 prefetch engines can be selected to fetch the second block of an L2 cache line, even if it is not required by the program. See [Section 3.6.3.2, “L2 Prefetch Engines and MSSCR0,”](#) for more information.

3.6.4.2 L2 Cache Allocation

Instruction cache misses in the L2 cache cause an L2 cache line to be allocated, provided the L2 cache is enabled and not marked as data-only (with the L2CR[L2DO] bit).

Also, data accesses cause an L2 cache line to be allocated if the L2 misses and the L2 is enabled and not marked as instruction-only (with the L2CR[L2IO] bit).

Write-back stores that miss in the L1 data cache but hit on an L2 cache block that is in the shared state are treated as store misses, causing a RWITM transaction to the MPX bus. In this case, the line is not deallocated, but it is reloaded as it is read from the MPX bus.

When the L1 data cache causes a castout and the L2 cache is enabled, the L2 cache does not allocate a new line for the castout if it misses. If the castout hits in the L2, the new castout data is written into the L2.

Transient accesses (caused by the **dstt**, **dststt**, **lvxl**, and **stvxl** instructions) are treated similarly to non-transient accesses, except that transient accesses do not cause entries to be allocated in the L2 cache on a miss. However, when an L1 data cache miss occurs for a transient operation, and the L2 cache hits, the L2 cache states are updated appropriately.

3.6.4.3 Store Data Merging and L2

Write-through stores use byte enables in the L1 data cache to merge the write data with the current cache contents (if it hits). For write-through stores ≥ 8 bytes that hit in the L2, the line is updated in the L2 with the store data, but the status bits for the block are not changed, and only the write-through store propagates to the MPX bus.

If a store of less than 8 bytes hits in the L2 cache,

- If the L2 block is modified, the block is cast out to the MPX bus, and only the write-through store propagates to the MPX bus
- If the L2 block is not modified, the block is invalidated, and only the write-through store propagates to the MPX bus.

3.6.4.4 L2 Cache Line Replacement Algorithms

The two pseudo-random modes of line replacement for the L2 cache (selected by L2CR[L2REP]) are three-bit counter mode and pseudo-random number generator mode. The three-bit counter mode (when L2CR[L2REP] = 1) is based on a simple three-bit counter that is incremented on every clock cycle. When a miss occurs, the line in the way pointed to by the counter is chosen for replacement.

The pseudo-random number generator mode (when L2CR[L2REP] = 0) uses 16 latches that are clocked on every clock cycle as shown in Figure 3-18 with three XOR functions. The L2 cache uses the value in latches 4, 9, and 15 as the 3-bit value that selects the way for replacement.

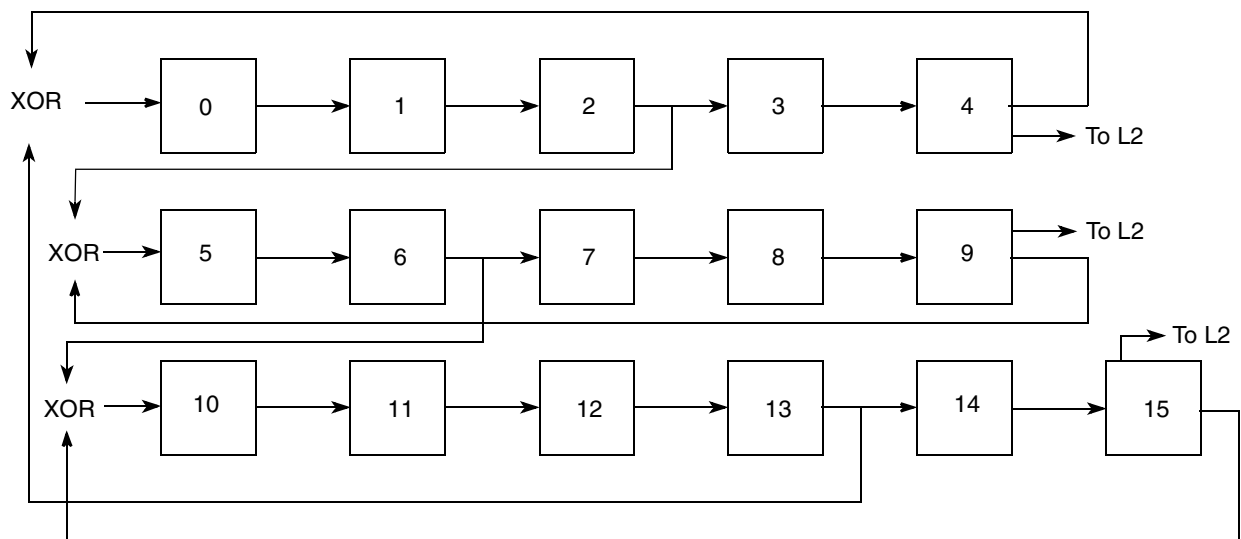


Figure 3-18. Random Number Generator for L2 Replacement Selection

Due to the latency of the L2 cache look-up, there are 3 clock cycles between a read miss and the allocation of the replacement line. Thus, it would be possible that the same way can be chosen for replacement for two, or even three consecutive read misses with the algorithm as described above. In order to avoid this, the actual algorithm compares a selected replacement line with the three previous replacement lines. If the selected line matches with one of the three previous ones, a value of 1, 2, or 3 is automatically added to the value that selects the way for replacement.

3.6.4.5 L2 Operations Caused by L1 Requests

This section contains a series of tables that define the actions of the L2 cache to service the L1 caches. See [Section 3.7.3.3, “L2 Operations Caused by Snoops External to the Core,”](#) for a description of L2 actions to service snoop requests.

[Table 3-14](#) through [Table 3-23](#) summarize all L2 cache activities and the internal conditions that cause them. [Table 3-13](#) defines some of the abbreviations used in [Table 3-14](#) through [Table 3-23](#). Note the following:

- The WIMG bits plus A (for atomic) are passed on to the core memory subsystem unless they are overridden.
- The t (transient) indicator is also passed on to the core memory subsystem.
- Any operation that requires an allocate in the L2 may fail to perform the allocate (whether due to a collision with a snoop, or due to the reload coming back faster than the allocate can arbitrate). In this case, the final state of the L2 cache will be the same as the initial state (I if invalid before, or S for store-hit_shared). For simplicity, the tables are written as if the allocate always succeeds.
- A **stwcx.** operation pending in the MSS that has not yet arbitrated or gone out on the MPX bus may lose its reservation while it is pending. If this occurs, the RWITM atomic transaction on the MPX bus is self-retried, and the operation is turned into a load operation. The MSS response to the L1 may be shared or exclusive depending on the bus response. For simplicity, these tables do not include that scenario, since it includes multiple transactions.

Table 3-13. Definitions for L2 Cache-State Summary

Term	Definition
L1 Snoop	The type of L1 snoop operation (if any) triggered by this MSS request.
MPX Bus Request	The MPX bus request (if any) triggered by this operation and its initial state. Any WIMG setting in the MPX bus request type is a forced value (MMU WIMG values are ignored).
Response on MPX Bus	The value of the shared snoop response (if applicable) to the MPX bus request.
Final L2 State	The MESI state of this address in the L2 cache after the operation completes. A represents the allocated state for retry conditions.
MSS Response to L1	If reloading the L1, whether the reload data is exclusive or shared.
SMC	Store miss complex. The series of queues that handle store misses.

Table 3-14. L2 Cache State Transitions for Load, lwarx, Touch, and lFetches

WIMG	Initial L2 State	MPX Bus Request	MPX Bus Response	Final L2 State	MSS Response to L1	Comments
l = 0 t = 0	I	Read W = 0	S	S	S	Forward the critical data to L1 (except touch). Reload L1 and L2 from MPX bus.
			E	E	E	
	S	None	N/A	Same	S	Reload L1 from L2.
E/M	None	N/A	Same	E		
l = 0 t = 1	I	Read W = 0	S	Same	S	Read or touch transient. Return reload data to L1, but don't allocate or reload L2.
			E	Same	E	
	S	None	N/A	Same	S	
	E/M	None	N/A	Same	E	
l = 1	N/A	Read W = 0	N/A	N/A	N/A	Bypass caches and perform cache-inhibited MPX bus read.

Table 3-15. L2 Cache State Transitions for Store Touch Operations

WIMG	Initial L2 State	MPX Bus Request	MPX Bus Response	Final L2 State	MSS Response to L1	Comments
l = 0 t = 0	I/S	RClaim	S	S	S	Reload L1 and L2 from MPX bus.
			E	E	E	
	E/M	None	N/A	Same	E	Reload L1 from L2.
l = 0 t = 1	I/S	RClaim	S	Same	S	Store touch transient. Return reload data to L1, but do not allocate or reload L2.
			E	Same	E	
	E/M	None	N/A	Same	E	

Table 3-16. L2 Cache State Transitions for Store (and stwcx.) Operations

WIMG	Initial L2 State	MPX Bus Request	MPX Bus Response	Final L2 State	MSS Response to L1	Comments
W = 0 l = 0 t = 0	I	RWITM	N/A	E	E	Reload L1 and L2 from MPX bus.
	S	RWITM	N/A	E	E	Reload L1/L2 from MPX bus.
	E/M	None	N/A	Same	E	Reload L1 from L2.
W = 0 l = 0 t = 1	I	RWITM	N/A	Same	E	Transient stores do not allocate/reload the L2 cache.
	S	RWITM	N/A	Same	E	If the L2 cache state gets flushed before reload, then the line stays invalid.
	E/M	None	N/A	Same	E	—

Table 3-16. L2 Cache State Transitions for Store (and stwcx.) Operations (continued)

WIMG	Initial L2 State	MPX Bus Request	MPX Bus Response	Final L2 State	MSS Response to L1	Comments
W = 1	I	Write with flush	N/A	Same	N/A	Do write-through store on MPX bus.
	S/E/M	Write with flush	N/A	Same	N/A	Merge data into L2. Put L2 data into L2SQ. Do write-through store of unmerged data on MPX bus.
I = 1	N/A	Write with flush	N/A	N/A	N/A	Bypass L2 cache and do cache-inhibited store on MPX bus.

Table 3-17. L2 Cache State Transitions for L1 Castout Operations

WIMG	Initial L2 State	MPX Bus Request	MPX Bus Response	Final L2 State	MSS Response to L1	Comments
W = 0 M = 0	I	Write w/Kill (W = 0, M = 0)	N/A	Same	N/A	Cast out L1 data to MPX bus.
	S/E/M	None	N/A	M	N/A	Cast out L1 data to L2.
W = 1	I/S/E/M	Write w/Kill (W = 1, M = 0)	N/A	I	N/A	Push data from L1 for dcbf .

Table 3-18. L2 Cache State Transitions for L2 Castout Operations

WIMG	Initial L2 State	MPX Bus Request	MPX Bus Response	Final L2 State	MSS Response to L1	Comments
x	M	Write w/Kill (W = 0, M = 0)	N/A	I	N/A	Cast out L2 data to MPX bus.

Table 3-19. L2 Cache State Transitions for dcbf Operations

WIMG	Initial L2 State	MPX Bus Request	MPX Bus Response	Final L2 State	MSS Response to L1	Comments
x	I	Flush	N/A	Same	N/A	—
	S/E	Flush	N/A	I	N/A	Invalidate L2.
	M	Write w/Kill (W = 1, M = 0)	N/A	I	N/A	Push data from L2 to MPX bus

Table 3-20. L2 Cache State Transitions for dcbz Operations

WIMG	Initial L2 State	MPX Bus Request	MPX Bus Response	Final L2 State	MSS Response to L1	Comments
M = 0	I/S/E/M	None	N/A	Same	E	No need to claim ownership. Synthesize L2 hit.

Table 3-20. L2 Cache State Transitions for dcbz Operations (continued)

WIMG	Initial L2 State	MPX Bus Request	MPX Bus Response	Final L2 State	MSS Response to L1	Comments
M = 1	I	Kill	N/A	E	E	Claim ownership for line.
	S	Kill	N/A	E	E	
	E/M	None	N/A	Same	E	—

Table 3-21. L2 Cache State Transitions for dcbst Operations

WIMG	Initial L2 State	MPX Bus Request	MPX Bus Response	Final L2 State	MSS Response to L1	Comments
x	I/S/E	Clean	N/A	Same	N/A	—
	M	Write w/Kill (W = 1, M = 0)	N/A	E	N/A	Push data from L2 to MPX bus.

Table 3-22. L2 Cache State Transitions for Write with Clean Operations

WIMG	Initial L2 State	MPX Bus Request	MPX Bus Response	Final L2 State	MSS Response to L1	Comments
x	I	Write w/Kill (W = 1, M = 0)	N/A	Same	N/A	Push data from L1 to MPX bus.
	S/E/M	Write w/Kill (W = 1, M = 0)	N/A	E	N/A	Push data from L1 to MPX bus, capturing it in L2.

Table 3-23. L2 Cache State Transitions for Remaining Instructions

MSS Op	Initial L2 State	MPX Bus Request	MPX Bus Response	Final L2 State	MSS Response to L1	Comments
icbi	N/A	ICBI	N/A	N/A	N/A	No action in L2 cache.
tlbie	N/A	TLBIE	N/A	N/A	N/A	No action in L2 cache.
tlbsync	N/A	TLBSYNC	N/A	N/A	N/A	No action in L2 cache
sync	N/A	SYNC	N/A	N/A	N/A	sync causes ordering of previous and subsequent loads/stores from the same processor.
eieio	N/A	EIEIO	N/A	N/A	N/A	eieio causes ordering of certain loads and stores.

3.7 Core Interface

The core interface buffers MPX bus requests from the L1 instruction cache, the L1 data cache, and the L2 cache, and executes the requests per the MPX bus protocol. The core interface captures snoop addresses for snooping in the caches, the address register queues, and the reservation address. For additional information about the e600 core interface, refer to [Chapter 8, “Core Interface.”](#)

3.7.1 MPX bus Operations Caused by Cache Control Instructions

The cache control, TLB management, and synchronization instructions supported by the e600 core may affect or be affected by operations on the MPX bus. The operation of the instructions may also indirectly cause MPX bus transactions to be performed, or their completion may be linked to actions on the MPX bus.

When memory coherency is required ($WIMG = xx1x$) and $HID1[ABE] = 1$, the **dcbst**, **dcbf**, **dcbi**, and **icbi** instructions are broadcast on the MPX bus as described in [Table 3-24](#). A **dcbi** or **dcbf** can create an address-only flush and a **dcbst** can create an address-only clean. When $M = 0$, **dcbst**, **dcbf**, and **dcbi** instructions are only broadcast on the MPX bus when the cache state hits as modified in either the L1 or L2 cache. Note that **dcbst**, **dcbf**, and **dcbi** instructions would create castout operations to the MPX bus if they hit modified within the caches even when $M = 0$. The **icbi** instruction is never broadcast when $M = 0$. A **dcbz** or **dcba** is never broadcast if the internal cache state is modified. For detailed information on the cache control instructions, refer to [Chapter 2, “Registers,”](#) in this book and Chapter 8, “Instruction Set,” in *The Programming Environments Manual*.

Table 3-24. MPX Bus Operations Caused by Cache Control Instructions ($WIMG = xx1x$)

Instruction	Current State		Next Cache State	MPX Bus Operation	Comment
	Cache Coherency	HID1 Setting			
sync	Do not care	SYNCBE = 0	No change	None	Waits for memory queues to complete MPX bus activity
		SYNCBE = 1		SYNC	
tlbie	Do not care	ABE = 0	No change	None	—
		ABE = 1		TLBIE	Address-only MPX bus operation
tlbsync	Do not care	ABE = 0	No change	None	—
		ABE = 1		TLBSYNC	Address-only MPX bus operation
eieio	Do not care	SYNCBE = 0	No change	None	—
		SYNCBE = 1		EIEIO	Address-only MPX bus operation
dcbt	M, E, S	—	No change	None	—
dcbt	I	—	E, S	Read	Fetches cache block is stored in the cache
dcbst	M, E, S	—	No change	None	—
dcbst	I	—	E, S	RCLAIM	Fetches cache block is stored in the cache
dcbz	M, E	—	M	None	Writes over modified data
dcbz	S, I	—	M	Kill	—
dcbst	M	—	E, S, I	Write with kill	—
dcbst	E, S, I	ABE = 0	No change or I	None	—
		ABE = 1		Clean	—
dcbf, dcbi	M	—	I	Write with kill	Block is pushed

Table 3-24. MPX Bus Operations Caused by Cache Control Instructions (WIMG = xx1x) (continued)

Instruction	Current State		Next Cache State	MPX Bus Operation	Comment
	Cache Coherency	HID1 Setting			
dcbf, dcbi	E, S, I	ABE = 0	I	None	—
		ABE = 1		Flush	Address-only bus operation
dcba	M, E	—	M	None	Writes over modified data
dcba	S, I	—	M	Kill	—
icbi	V, I	ABE = 0	I	None	Instruction cache only
		ABE = 1		ICBI	Instruction cache only

When memory coherency is not required (WIMG = xx0x), the **dcbz**, **dcba**, **dcbf**, **dcbi**, and **dcbtst** instructions are broadcast on the MPX bus as described in [Table 3-25](#).

Table 3-25. MPX Bus Operations Caused by Cache Control Instructions (WIMG = xx0x)

Instruction	Current State		Next Cache State	MPX Bus Operation	Comment
	Cache Coherency	HID1 Setting			
dcbt	M, E, S	—	No change	None	—
dcbt	I	—	E, S	Read	Fetches cache block is stored in the cache
dcbtst	M, E, S	—	No change	None	—
dcbtst	I	—	E, S	RCLAIM	Fetches cache block is stored in the cache
dcbz	M, E, S, I	—	M	None	
dcba	M, E, S, I	—	M	None	
dcbf, dcbi	M	—	I	Write with kill	Block is pushed
dcbf, dcbi	E, S, I	—	I	None	—
dcbst	M	—	E, S, I	Write with kill	Block is pushed
dcbst	E, S, I	—	I	None	—
icbi	V, I	—	I	None	Instruction cache only

3.7.2 Transfer Attributes

In addition to the address and transfer type signals, the e600 core supports the transfer attribute signals \overline{wt} , \overline{ci} , and \overline{gbl} .

The \overline{wt} signal reflects the write-through/write-back status (the complement of the W bit) for the transaction as determined by the MMU address translation during write operations. \overline{wt} is also asserted when the data

cache is locked (with $HID0[DLOCK]$ or $LDSTCR[DCWL] = 0xFF$) and for burst writes due to **dcbf** (flush) and **dcbst** (clean) instructions and snoop pushes.

For read transactions, the \overline{wt} signal reflects whether the access is an instruction or data access as follows:

- \overline{wt} is asserted for data reads
- \overline{wt} is negated for instruction reads

The \overline{ci} signal reflects the caching-inhibited/caching-allowed status (the complement of the I bit) of the transaction as determined by the MMU address translation. The \overline{ci} signal is asserted for data loads or stores if the L1 data cache is disabled.

The \overline{gbl} signal reflects the memory coherency requirements (the complement of the M bit) of the transaction as determined by the MMU address translation. Address bus masters assert \overline{gbl} to indicate that the current transaction is a global access (that is, an access to memory shared by more than one device). Because cache block castouts and snoop pushes do not require snooping, the \overline{gbl} signal is not asserted for these operations. Note that \overline{gbl} is asserted for all data read or write operations when using real addressing mode (that is, address translation is disabled).

Table 3-26 summarizes the address and transfer attribute information presented on the MPX bus by the e600 core for various master or snoop-related transactions. Note that the address ranges shown in the table apply when 36-bit physical addressing is used ($HID0[XAEN] = 1$).

Table 3-26. Address/Transfer Attributes Generated by the e600 Core

MPX Bus Transaction	Addr [0:35]	$tt[0:4]$	\overline{wt}	\overline{ci}	\overline{gbl}
Instruction fetch operations					
Burst	PA[0:32] 0b000	0 1 0 1 0	1	1	¬ M
Data cache operations					
Cache block fill (due to load miss)	PA[0:32] 0b000	A 1 0 1 0	0	1	¬ M
Cache block fill (due to store miss)	PA[0:32] 0b000	A 1 1 1 0	1	1	¬ M
Store merged to 32 bytes	PA[0:30] 0b00000	0 1 1 0 0	1	1	¬ M
Castout (normal replacement)	CA[0:30] 0b00000	0 0 1 1 0	1	1	1
Cache block clean due to dcbst hit to modified	PA[0:30] 0b00000	0 0 1 1 0	0	1	1
Cache block flush due to dcbf hit to modified	PA[0:30] 0b00000	0 0 1 1 0	0	1	1
Snoop copyback	CA[0:30] 0b00000	0 0 1 1 0	0	1	1
dcbt, dst, dstt	PA[0:30] 0b00000	0 1 0 1 0	0	1	¬ M
dcbst, dstst, dststt	PA[0:30] 0b00000	0 1 1 1 1	0	1	¬ M

Table 3-26. Address/Transfer Attributes Generated by the e600 Core (continued)

MPX Bus Transaction	Addr [0:35]	<i>tt</i> [0:4]	\overline{wt}	\overline{ci}	\overline{gbl}
Data cache bypass operations					
Single-beat read (caching-inhibited or cache disabled)	PA[0:35]	0 1 0 1 0	0	$\neg I$	$\neg M$
Altivec load (caching-inhibited, write-through, or cache disabled)	PA[0:32] 0b000	0 1 0 1 0	0	$\neg I$	$\neg M$
Single-beat write (caching-inhibited, write-through, cache disabled, or cache completely locked)	PA[0:35]	0 0 0 1 0	$\neg W$	$\neg I$	$\neg M$
Special instructions					
icbi (addr-only)	PA[0:30] 0b00000	0 1 1 0 1	$\neg W$	$\neg I$	$\neg M$
dcba (addr-only)	PA[0:30] 0b00000	0 1 1 0 0	1	1	0
dcbz (addr-only)	PA[0:30] 0b00000	0 1 1 0 0	1	1	0
dcbf, dcbi (addr-only)	PA[0:30] 0b00000	0 0 1 0 0	$\neg W$	$\neg I$	$\neg M$
dcbst (addr-only)	PA[0:30] 0b00000	0 0 0 0 0	$\neg W$	$\neg I$	$\neg M$
sync (addr-only)	0x0_0000_0000	0 1 0 0 0	1	1	0
tlbsync (addr-only)	0x0_0000_0000	0 1 0 0 1	1	1	0
tlbie (addr-only)	0b0000 EA[0:31]	1 1 0 0 0	1	1	0
eieio (addr-only)	0x0_0000_0000	1 0 0 0 0	1	1	0

3.7.3 Snooping of MPX Bus Transactions

The e600 core maintains data cache coherency in hardware by coordinating activity between the data cache, the L2 cache, the core memory subsystem, and devices on the MPX bus. The core has a write-back caching capability that relies on bus snooping to maintain cache coherency with other caches in the system. For the e600 core, the coherency size of the MPX bus is 32 bytes, the size of a cache block. This means that any MPX bus transactions that cross an aligned 32-byte boundary must present a new address onto the MPX bus at that boundary for proper snoop operation by the e600 core, or they must operate noncoherently with respect to the e600 core.

As bus operations are performed on the MPX bus by other bus masters, the e600 core bus snooping logic monitors the addresses and transfer attributes that are referenced. The core must see all system coherency snoops to function properly in a symmetric multi-processing (SMP) environment. The e600 core cannot support devices external to the core that filter out snoop traffic on the MPX bus (for example, an external, in-line cache).

The e600 core snoops bus transactions during the transfer start clock all global transactions (\overline{gbl} asserted).

3.7.3.1 Types of Transactions Snooped by the e600 Core

There are several bus transaction types defined for the MPX bus. As shown in [Table 3-27](#), the e600 core snoops many, but not all, MPX bus transactions. The transactions in [Table 3-27](#) correspond to the transfer type signals *tt*[0:4].

Table 3-27. Snooped Bus Transaction Summary

Transaction	tt[0:4]	Snooped by e600
Clean	00000	Yes
Flush	00100	Yes
sync	01000	Yes
Kill	01100	Yes
eieio	10000	No
External control word write	10100	No
TLB invalidate	11000	Yes
External control word read	11100	No
lwarx reservation set	00001	No
Reserved	00101	No
tlbsync	01001	Yes
icbi	01101	Yes
Reserved	1XX01	No
Write-with-flush	00010	Yes
Write-with-kill	00110	Yes
Read	01010	Yes
Read-with-intent-to-modify (RWITM)	01110	Yes
Write-with-flush-atomic	10010	Yes
Reserved	10110	No
Read-atomic	11010	Yes
Read-with-intent-to-modify-atomic	11110	Yes
Reserved	00011	No
Reserved	00111	No
Read-with-no-intent-to-cache (RWNITC)	01011	Yes
Read-claim (RCLAIM)	01111	Yes
Reserved	1XX11	No

Once a qualified snoop condition is detected on the MPX bus, the snooped address is compared against the data cache tags, the LSU and core memory subsystem queues, reservation address, and/or other storage elements as appropriate. The L1 data cache tags and L2 cache tags are snooped for standard data cache coherency support. No snooping is done in the instruction cache for coherency (except that the **icbi** instruction can cause matching entries to be invalidated).

The LSU and core memory subsystem queues are snooped for pipeline collisions and memory coherency collisions. A pipeline collision is detected when another bus master addresses any portion of a line that this

e600 core is currently processing in its caches. A memory coherency collision occurs when another bus master addresses any portion of a line that the core has currently queued to write to memory from the data cache (castout or push), but has not yet been granted bus access to perform.

If the snooped address does not hit in the cache, snooping finishes with no action taken. If, however, the address hits in the cache, the e600 core reacts according to the coherency protocol diagrams shown in [Section 3.3.2.5, “MESI State Transitions.”](#)

3.7.3.2 L1 Cache State Transitions and Bus Operations Due to Snoops

[Table 3-30](#) shows the state transitions in the L1 caches for each snoop type. For each snoop, the L1 responds with valid if the line was shared, exclusive, or modified, and modified if the line was modified. The snoop types in [Table 3-30](#) are listed in [Table 3-28](#).

Table 3-28. Definitions of Snoop Type for L1 Cache/Snoop Summary

Snoop Type	Definition
Snoop Kill Reservation	If the snoop address matches a valid reservation in the core, kill the reservation if there is no retry. This operation is caused by RWITM, RWITM ATOMIC, RCLAIM, and KILL snoops external to the core.
Snoop Flush-Kill	Push any modified data to the L1 push buffer and invalidate the line. Return the initial MESI state of the line. This operation is caused by stores with I = 0 and W = 0, stwcx with W = 0 and I = 0, dcbstst , dstst , dcbz , or dcba instructions within the core.
Snoop Flush	Push any modified data to the L1 push buffer and invalidate the line. Return the initial MESI state of the line. This operation is caused by dcbf instruction within the core, or RWITM, RWITM ATOMIC, RCLAIM, KILL, WRITE W/FLUSH, WRITE W/FLUSH ATOMIC, or FLUSH snoops external to the core.
Snoop Read	Push any modified data to the L1 push buffer. If the line was valid, leave it shared. Return the initial MESI state of the line. This operation is caused by dcbst , load, lwarx or touch instructions within the core or CLEAN, RWNITC, READ, READ ATOMIC snoops external to the core.
Snoop icbi	Invalidate the line in the instruction cache. This operation is caused by an icbi instruction within the core or an ICBI external snoop.
Snoop tlbie	Invalidate all matching PTEs in the instruction and data TLBs. Mark all outstanding memory accesses that used old translations. This operation is caused by a TLBIE external snoop.

[Table 3-29](#) defines some terms used in [Table 3-30](#).

Table 3-29. Definitions of Other Terms for L1 Cache/Snoop Summary

Term	Definition
Snoop Type	The local snoop type. See Section 3.6.4.5, “L2 Operations Caused by L1 Requests,” and Section 3.7.3.3, “L2 Operations Caused by Snoops External to the Core,” for a list of the operations and states that cause the various L1 snoop types.
Initial L1 State	The MESI state of the cache before the snoop begins.
Final L1 State	If the L1 MESI state is unchanged, then the entry is marked as same; otherwise, the MESI state at the end of the snoop operations.

Table 3-30. L1 Cache State Transitions Due to Snoops

Snoop Type	Initial L1 State	Final L1 State	Action	Comments
Kill Reservation	N/A	Same	Kill Reservation if the address matches and no core retried the operation.	—
Flush-Kill	I	Same	If L1 = M, data is moved to push buffer. MSS will request it. Kill Reservation if the address matches and no core retried the operation.	Cache line is invalidated.
	S/E/M	I		
Flush	I	Same	If L1 = M, data is moved to push buffer. MSS will request it.	Cache line is invalidated.
	S/E/M	I		
Read	I/S	Same	If L1 = M, data is moved to push buffer. MSS will request it.	If cache line was valid, leave it shared. Note: if the reservation address matched but the cache line was invalid, the L1 snoop logic will synthesize a shared response to the LMQ or the MPX bus.
	E/M	S		
icbi	N/A	N/A	Invalidate matching line in the instruction cache.	—
tlbie	N/A	N/A	Invalidate matching TLB entries in the ITLB and DTLB.	—

3.7.3.3 L2 Operations Caused by Snoops External to the Core

The L1 and L2 cache states affect the response to snoops external to the core. Some snoop types do not affect the caches. Table 3-32 shows the response to all snooped MPX bus operations, depending on the initial cache state. See [Section 3.5.9, “L1 Cache Operation Summary,”](#) for a description of the L1 snoop responses.

[Table 3-32](#) shows the state transitions in the L1 and L2 caches for each external snoop operation. The table lists only legal state combinations. The columns are defined in [Table 3-31](#).

Table 3-31. Definitions for L2 Cache/Snoop Summary

Term	Definition
Snoop Type	The MPX bus transfer type as described in Table 3-30.
L1 Snoop Type	The L1 snoop operation (if any) triggered by this operation.
L1 Response	The invalid/valid/modified response of the core. If the L1 response is retry, the MPX bus response is always retry.
Initial L2 State	The MESI state of this address in the L2 cache before the snoop operation.
Final L2 State	The MESI state of this address in the L2 cache when all operations triggered by this snoop are complete.
MPX Bus Response	Shared indicates there is a valid copy of the data and the data stays valid (if the MPX bus operation supports shared response). Modified indicates there is a modified copy of the data and the cache will provide intervention data. Retry indicates the master must try the operation again to get the most up-to-date data and a clean response. Shared and retry together indicates the this device must perform a push. Shared and modified together indicates this device will provide intervention data and retain a valid copy of the line.

Note the following:

- Snoop kill reservation is performed only if a matching reservation exists in the L1 (part of the L1 snoop response).
- For write-with-kill and kill snoops external to the core, a flush operation is sent to the L1's. If the L1 is modified, a push is generated in the SMC (store miss complex). The external snoop logic and SMC contain circuitry to drop this push if the snoop is not retried on the MPX bus. The e600 core does not respond retry if there is modified data in the L1 or L2.
- The atomic bus operations have the same snoop responses as the non-atomic ones.
- Because the e600 core only snoops global accesses (\overline{gbl} asserted), that is assumed for all of the tables. The e600 core will not issue a snoop response for transactions in which gbl is not asserted.

Table 3-32. External Snoop Responses and L1 and L2 Actions

Snoop Type	L1 Snoop Type	L1 Response	Initial L2 State	Final L2 State	Response to Snoop	Comments
Flush	Flush	I/V	I//S/E	I	None	Invalidate L1/L2.
		M	I/S/E	I	GBL	Intervene from L1 and invalidate L1/L2.
Write with flush	Flush ¹	I/V	I/S/E	I	None	Invalidate L1.
			M	I	Retry	Cache paradox: Push block from L2. Invalidate L1/L2.
		M	I/S/E/M	I	Retry	Cache paradox: Push block from L1. Invalidate L1.
Kill	Flush	I/V/M	I/S/E/M	I	None	Invalidate L1/L2. Kill reservation (if necessary). If L1 response is M, the push in the SMC is killed if MPX bus response is not retry.
	Kill Reservation					
Write with kill	Flush	I/V/M	I/S/E/M	I	None	Invalidate L1/L2. If L1 response is M, the push in the SMC is killed if MPX bus response is not retry.
Read	Read	I/V	I	Same	None	—
			S/E	S	Shared	Set L1/L2 shared.
			M	S	Shared + GBL	Intervene from L2, and set L1/L2 shared.
		M	I	Same	Shared + GBL	Intervene from L1, consume data in L2 if valid, and set L1/L2 shared.
			S/E	S		
			M	S		

Table 3-32. External Snoop Responses and L1 and L2 Actions (continued)

Snoop Type	L1 Snoop Type	L1 Response	Initial L2 State	Final L2 State	Response to Snoop	Comments
Clean RWNITC	Read	I/V	I	Same	None	—
			S/E	Same	None	Clean L1/L2.
			M	E	GBL	Intervene from L2 and clean L1/L2.
		M	I	Same	GBL	Intervene from L1 and clean L1/L2. Consume intervention data in L2 if valid.
			S/E	S		
			M	S		
RWITM RCLAIM	Flush Kill Reservation	I/S/E	I/S/E	I	None	Invalidate L1. Kill reservations (if necessary).
			M	I	GBL	Intervene from L2. Invalidate L2/L2. Kill reservation (if necessary).
		M	I/S/E	I	GBL	Intervene from L1. Invalidate L2/L2. Kill reservation (if necessary).
TLBIE	tlbie	N/A	N/A	N/A	None	Snoop core only, no L2 action.
ICBI	icbi	N/A	N/A	N/A	None	
SYNC	None	N/A	N/A	N/A	None	The sync instruction does not need to query.
TLBSYNC	None	N/A	N/A	N/A	None	The tlbsync instruction does not need to query. It asserts a retry if and only if there is a pending marked transaction from a previous tlbie .
EIEIO	N/A	N/A	N/A	N/A	None	e600 does not snoop eieio .
LWARX RESERVE	N/A	N/A	N/A	N/A	None	e600 does not snoop lwarx reserve.
xferdata	N/A	N/A	N/A	N/A	None	e600 does not snoop xferdata in or out.

¹ Snoop W = 1 or I = 1 Write w/Flush need not kill reservations because **lwarx** is not supported in W = 1 or I = 1 space, and aliasing W = 1 and W = 0 or I = 1 and I = 0 across processors is illegal.



Chapter 4 Interrupts

The OEA portion of the PowerPC architecture defines the mechanism by which processors implement interrupts. Exception conditions, the events that when encountered may result in an interrupt, may be defined at other levels of the architecture or may be implementation dependent. For example, the UISA defines conditions that may cause floating-point exceptions; the OEA defines the mechanism by which the interrupt is taken.

The interrupt mechanism allows the processor that implements the PowerPC architecture to change to supervisor state as a result of unusual conditions arising in the execution of instructions and from signals external to the core, MPX bus errors, or various internal conditions. When interrupts occur, information about the state of the processor is saved to certain registers and the processor begins execution at an address (interrupt vector) predetermined for each interrupt. Processing of interrupts begins in supervisor mode.

Although multiple exception conditions can map to a single interrupt vector, often a more specific condition may be determined by examining a register associated with the interrupt—for example, the DSISR and the floating-point status and control register (FPSCR). Also, software can explicitly enable or disable some exception conditions.

The PowerPC architecture requires that interrupts be taken in program order; therefore, although a particular implementation may recognize interrupt conditions out of order, they are handled strictly in order with respect to the instruction stream. When an instruction-caused interrupt is recognized, any unexecuted instructions that appear earlier in the instruction stream, including any that have not yet entered the execute state, are required to complete before the interrupt is taken. In addition, if a single instruction encounters multiple exception conditions, those interrupts are taken and handled sequentially. Likewise, asynchronous, precise interrupts are recognized when they occur, but are not handled until all instructions currently in the execute stage successfully complete execution and report their results. To prevent loss of state information, interrupt handlers must save the information stored in the machine status save/restore registers, SRR0 and SRR1, soon after the interrupt is taken to prevent this information from being lost due to another interrupt being taken. Because interrupts can occur while an interrupt handler routine is executing, multiple interrupts can become nested. It is up to the interrupt handler to save the necessary state information if control is to return to the excepting program.

In many cases, after the interrupt handler handles an interrupt, there is an attempt to execute the instruction that caused the interrupt. Instruction execution continues until the next interrupt condition is encountered. Recognizing and handling interrupt conditions sequentially guarantees that the machine state is recoverable and processing can resume without losing instruction results.

Interrupts

In this book, the following terms are used to describe the stages of interrupt processing:

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

AltiVec Technology and the Interrupt Model

Only the four following interrupts may result from execution of an AltiVec instruction:

- An AltiVec unavailable interrupt occurs with an attempt to execute any non-stream AltiVec instruction with $MSR[VEC] = 0$. After this interrupt occurs, execution resumes at offset 0x00F20 from the physical base address indicated by $MSR[IP]$. This interrupt does not occur for data streaming instructions (**dst[t]**, **dstst[t]**, **dss**, and **dssall**). Also note that the $VRSAVE$ register is not protected by this interrupt; this is consistent with the *AltiVec Programming Environments Manual*.
- A DSI interrupt occurs for an AltiVec load or store only if the load or store operation encounters a page fault (does not find a valid PTE during a table search operation) or a protection violation. Also a DSI interrupt occurs if an AltiVec load or store attempts to access a $SR[T] = 1$ (direct-store) memory location.
- An AltiVec assist interrupt may occur if an AltiVec floating-point instruction detects denormalized data as an input or output in Java mode. After this interrupt occurs, execution resumes at offset 0x01600 from the physical base address indicated by $MSR[IP]$.
- AltiVec loads and stores

4.1 e600 Core Interrupts

As specified by the PowerPC architecture, interrupts can be either precise or imprecise and either synchronous or asynchronous. Asynchronous interrupts are caused by events external to the core's execution; synchronous interrupts are caused by instructions.

The types of interrupts are shown in [Table 4-1](#). Note that all interrupts except for the performance monitor, AltiVec unavailable, instruction address breakpoint, system management, AltiVec assist, and the 3 software table search interrupts are described in Chapter 6, "Interrupts," in *The Programming Environments Manual*.

Table 4-1. e600 Core Interrupt Classifications

Synchronous/Asynchronous	Precise/Imprecise	Interrupt Types
Asynchronous, nonmaskable	Imprecise	System reset, machine check
Asynchronous, maskable	Precise	External interrupt, system management interrupt, decremter interrupt, performance monitor interrupt
Synchronous	Precise	Instruction-caused interrupts

The interrupt classifications are discussed in greater detail in [Section 4.2, "e600 Core Interrupt Recognition and Priorities."](#) For a better understanding of how the e600 core implements precise interrupts, see [Chapter 6, "Instruction Timing."](#) Interrupts implemented in the e600, and conditions that cause them, are listed in [Table 4-2](#). [Table 4-2](#) notes when an interrupt is implementation-specific to the e600 core. The three software table search interrupts are used by the core when $HID0[STEN] = 1$, to support the software page table searching. Refer to [Section 4.6.15, "TLB Miss Interrupts,"](#) and [Chapter 5, "Memory Management,"](#) for more information about the software table search operations.

Table 4-2. Interrupts and Exception Conditions

Interrupt	Vector Offset	Exception Conditions
Reserved	0x00000	—
System reset	0x00100	Assertion of either \overline{hreset} or \overline{sreset} or at power-on reset
Machine check	0x00200	Assertion of \overline{tea} during a data bus transaction, assertion of \overline{mcp} , an address bus parity error on the MPX bus, a data bus parity error on the MPX bus, an L1 instruction cache error, and L1 data cache error, a core memory subsystem detected error including the following: <ul style="list-style-type: none"> • L2 data parity error • L2 tag parity error • Single-bit and multiple-bit L2 ECC errors MSR[ME] must be set.
DSI	0x00300	As specified in the PowerPC architecture. Also includes the following: <ul style="list-style-type: none"> • A hardware table search due to a TLB miss on load, store, or cache operations results in a page fault. • Any load or store to a direct-store segment ($SR[T] = 1$). • A lwarx or stwcx. instruction to memory with cache-inhibited or write-through memory/cache access attributes.
ISI	0x00400	As specified in the PowerPC architecture
External interrupt	0x00500	$MSR[EE] = 1$ and \overline{int} is asserted

Table 4-2. Interrupts and Exception Conditions (continued)

Interrupt	Vector Offset	Exception Conditions
Alignment	0x00600	<ul style="list-style-type: none"> A floating-point load/store, stmw, stwcx., lmw, or lwarx instruction operand is not word-aligned. A multiple/string load/store operation is attempted in little-endian mode An operand of a dcbz instruction is on a page that is write-through or cache-inhibited for a virtual mode access. An attempt to execute a dcbz instruction occurs when the cache is disabled or locked.
Program	0x00700	As specified in the PowerPC architecture
Floating-point unavailable	0x00800	As specified in the PowerPC architecture
Decrementer	0x00900	As defined by the PowerPC architecture, when the msb of the DEC register changes from 0 to 1 and MSR[EE] = 1.
Reserved	0x00A00–00BFF	—
System call	0x00C00	Execution of the System Call (sc) instruction
Trace	0x00D00	MSR[SE] = 1 or a branch instruction is completing and MSR[BE] = 1. The e600 core operates as specified in the OEA by taking this interrupt on an isync .
Reserved	0x00E00	The e600 core does not generate an interrupt to this vector. Other processors may use this vector for floating-point assist interrupts.
Reserved	0x00E10–00EFF	—
Performance monitor	0x00F00	The limit specified in PMC <i>n</i> is met and MMCR0[ENINT] = 1 (e600-specific).
AltiVec unavailable	0x00F20	Occurs due to an attempt to execute any non-streaming AltiVec instruction when MSR[VEC] = 0. This interrupt is not taken for data streaming instructions (dstx , dss , or dssall) (e600-specific).
ITLB miss	0x01000	Caused when HID0[STEN] = 1 and the effective address for an instruction fetch cannot be translated by the ITLB (e600-specific).
DTLB miss-on-load	0x01100	Caused when HID0[STEN] = 1 and the effective address for a data load operation cannot be translated by the DTLB (e600-specific).
DTLB miss-on-store	0x01200	Caused when HID0[STEN] = 1 and the effective address for a data store operation cannot be translated by the DTLB, or when a DTLB hit occurs, and the changed bit in the PTE must be set due to a data store operation (e600-specific).
Instruction address breakpoint	0x01300	IABR[0–29] matches EA[0–29] of the next instruction to complete and IABR[BE] = 1 (e600-specific).
System management	0x01400	MSR[EE] = 1 and \overline{smi} is asserted (e600-specific).
Reserved	0x01500–015FF	—
AltiVec assist	0x01600	This e600-specific exception supports denormalization detection in Java mode as specified in the <i>AltiVec Technology Programming Environments Manual</i> .
Reserved	0x01700–02FFF	—

4.2 e600 Core Interrupt Recognition and Priorities

Interrupts are roughly prioritized by class, as follows:

1. Nonmaskable, asynchronous interrupts such as system reset and machine check interrupts, have priority over all other interrupts although the machine check interrupt condition can be disabled so the condition causes the processor to go directly into the checkstop state. These interrupts cannot be delayed and do not wait for completion of any precise interrupt handling.
2. Synchronous, precise interrupts are caused by instructions and are taken in strict program order.
3. Imprecise interrupts (imprecise mode floating-point enabled interrupts) are caused by instructions and they are delayed until higher priority interrupts are taken. Note that the e600 core does not implement an interrupt of this type.
4. Maskable asynchronous interrupts (external interrupt, decremter, system management interrupt, and performance monitor interrupts) are delayed until higher priority interrupts are taken.

The following list of categories describes how the e600 core handles interrupt conditions up to the point that the interrupt is taken. Note that a recoverable state is reached if the completed store queue is empty and any instruction that is next in program order, and has been signaled to complete, has completed. If $MSR[RI] = 0$, the core is in a nonrecoverable state. Also, instruction completion is defined as updating all architectural registers associated with that instruction, and then removing that instruction from the completion buffer. When all the pending store instructions have been committed to memory, the completed store queue is empty.

- Interrupts caused by asynchronous events (interrupts). These interrupts are further distinguished by whether they are maskable and recoverable.
 - Asynchronous, nonmaskable, nonrecoverable
System reset for assertion of \overline{hreset} —Has highest priority and is taken immediately regardless of other pending interrupts or recoverability (includes power-on reset).
 - Asynchronous, maskable, nonrecoverable
Machine check interrupt—Has priority over any other pending interrupt except system reset for assertion of \overline{hreset} (or power-on reset). Taken immediately regardless of recoverability.
 - Asynchronous, nonmaskable, recoverable
System reset for \overline{sreset} —Has priority over any other pending interrupt except system reset for \overline{hreset} (or power-on reset), or machine check. Taken immediately when a recoverable state is reached.
 - Asynchronous, maskable, recoverable
System management interrupt, performance monitor, external interrupt, and decremter interrupts—Before handling this type of interrupt, the next instruction in program order must complete. If that instruction causes another type of interrupt, that interrupt is taken and the asynchronous, maskable recoverable interrupt remains pending until the instruction completes. Further instruction completion is halted. The asynchronous, maskable recoverable interrupt is taken when a recoverable state is reached.

- Instruction-related interrupts. These interrupts are further organized into the point in instruction processing at which they generate an interrupt.
 - Instruction fetch and ITLB miss
 - ISI interrupts—Once this type of interrupt is detected, fetching stops and the current instruction stream is allowed to drain out of the machine. If completing any of the instructions in this stream causes an interrupt, that interrupt is taken and the instruction fetch interrupt is discarded, but may be encountered again when instruction processing resumes. Otherwise, once all pending instructions have executed and a recoverable state is reached, the ISI or ITLB miss interrupt is taken.
 - Instruction dispatch/execution
 - Program, DSI, alignment, floating-point unavailable, AltiVec unavailable, AltiVec assist, system call, instruction address breakpoint, data address breakpoint, and DTLB miss (if $HID0[STEN] = 1$)—This type of interrupt is determined during dispatch or execution of an instruction. The interrupt remains pending until all instructions before the interrupt-causing instruction in program order complete. The interrupt is then taken without completing the interrupt-causing instruction. If completing these previous instructions causes an interrupt, that interrupt takes priority over the pending instruction dispatch/execution interrupt, which is discarded, but may be encountered again when instruction processing resumes.
 - Post-instruction execution
 - Trace—Trace interrupts are generated following execution and completion of an instruction while trace mode is enabled. If executing the instruction produces conditions for another type of interrupt, that interrupt is taken and the post-instruction interrupt is ignored for that instruction.

Note that these interrupt classifications correspond to how interrupts are prioritized, as described in [Table 4-3](#).

Table 4-3. e600 Interrupt Priorities

Priority	Exception	Cause
Asynchronous Interrupts		
0	System reset	Power-on reset, assertion of \overline{hreset} and \overline{trst} (hard reset)
1	Machine check	Any enabled machine check condition (assertion of \overline{tea} or \overline{mcp} , or core memory subsystem error as defined in MSSSR0 (See Section 2.1.6.4, “Memory Subsystem Status Register (MSSSR0).”), address or data parity error, L1 address or data parity error, data cache error, instruction cache error, L2 data parity error, L2 tag error, L2 tag parity error, single-bit and multiple-bit L2 ECC error if ECC is enabled)
2	System reset	Assertion of \overline{sreset} (soft reset)
3	System management interrupt	Assertion of \overline{smi}
4	External interrupt	Assertion of \overline{int}
5	Performance monitor	Any programmer-specified performance monitor condition
6	Decrementer	Decrementer passes through zero.

Table 4-3. e600 Interrupt Priorities (continued)

Priority	Exception	Cause
Instruction Fetch Interrupts		
0	ISI	ISI interrupt conditions due to one of the following: 1. No-execute segment 2. Direct-store (T=1) segment
1	ITLB Miss	Instruction table miss interrupt due to miss in ITLB with HID0[STEN] = 1
2	ISI	ISI interrupt conditions due to one of the following: 1. Effective address can not be translated (page fault) 2. Instruction fetch from guarded memory 3. Protection violation
Instruction Dispatch/Execution Interrupts		
0	Instruction address breakpoint	Any instruction address breakpoint exception condition
1	Program	Illegal instruction, privileged instruction, or trap exception condition. Note that floating-point enabled program exceptions have lower priority.
2	System call	System call (sc) instruction
3	Floating-point unavailable	Any floating-point unavailable exception condition
4	AltiVec unavailable	Any unavailable AltiVec interrupt condition
5	Program	A floating-point enabled exception (lowest-priority program interrupt)
6	Alignment	Any alignment interrupt condition, prioritized as follows: 1. Floating-point access not word-aligned 2. lmw , stmw , lwarx , or stwcx . not word-aligned 3. Multiple or string access with MSR[LE] set 4. dcbz to a locked or disabled L1 data cache, WT, or CI page 5. stvx , stvxl , lvx , or lvxl to a disabled L1 cache
7	DSI	DSI interrupt due to execution of stvx , stvxl , lvx , or lvxl with all of the following conditions: • SR[T] = 0 (with BAT miss) • Cache-inhibited or write-through space
8	Alignment	Alignment interrupt due to execution of stvx , stvxl , lvx , or lvxl with all of the following conditions: • SR[T]=1(with BAT miss) • Cache-inhibited or write-through space
9	DSI	DSI interrupt due to eciwx or ecowx with EAR[E] = 0 (DSISR[11]).
10	DSI	DSI interrupt due to lwarx/stwcx . with caching disabled or if all ways are locked.
11	DSI	DSI interrupt due to the following: • BAT/page protection violation (DSISR[4]), or • lwarx/stwcx . to BAT entry with write-through attributes (W = 1), or to a page table entry (or BAT entry) with caching disallowed attributes (I = 1), or to a page table entry (or BAT entry) with caching-allowed attributes (I = 0), but with a locked L1 data cache (DSISR[5]) Note that if both occur simultaneously, both bits 4 and 5 of the DSISR are set.

Table 4-3. e600 Interrupt Priorities (continued)

Priority	Exception	Cause
12	DSI	DSI interrupt due to any access except cache operations to a segment where SR[T] = 1 (DSISR[5]) or an access crosses from a T = 0 segment to one where T = 1 (DSISR[5])
13	DTLB miss on store	Data table miss on store interrupt due to store miss in DTLB with HID0[STEN] = 1
14	DTLB miss-on-load	Data table miss-on-load interrupt due to load miss in DTLB with HID0[STEN] = 1
15	DSI	DSI interrupt due to one of the following: <ul style="list-style-type: none"> • TLB translation detects page protection violation (DSISR[4]) • TLB translation detects lwarx/stwcx. to a page table entry with write-through attributes (W = 1), or to a page table entry (or BAT entry) with caching disallowed attributes (I = 1), or to a page table entry (or BAT entry) with caching-allowed attributes (I = 0), but with a locked L1 data cache (DSISR[5]). • Hardware table search page fault (DSISR[1]) Note that if both 1 and 2 occur simultaneously, both bits 4 and 5 of the DSISR are set.
16	DTLB miss on store (data store access and C = 0)	Data TLB miss on store interrupt when HID0[STEN] = 1 and the PTE changed bit is not set (C = 0) for a store operation.
17	DSI	DSI interrupt due to DABR address match (DSISR[9]). Note that even though DSISR[5] and DSISR[9] are set by interrupts with different priorities, they can be set simultaneously.
18	AltiVec assist	Denormalized data detected as input or output in the AltiVec vector floating-point unit (VFPU) while in Java mode (VSCR[NJ] = 0)
Post-Instruction Execution Interrupts		
19	Trace	MSR[SE] = 1 (or MSR[BE] = 1 for branches)

System reset and machine check interrupts may occur at any time and are not delayed even if an interrupt is being handled. As a result, state information for an interrupted interrupt may be lost; therefore, these interrupts are typically nonrecoverable. An interrupt may not be taken immediately when it is recognized.

4.3 Interrupt Processing

When an interrupt is taken, the processor uses SRR0 and SRR1 to save the contents of the MSR for the current context and to identify where instruction execution should resume after the interrupt is handled.

When an interrupt occurs, the address saved in SRR0 helps determine where instruction processing should resume when the interrupt handler returns control to the interrupted process. Depending on the interrupt, this may be the address in SRR0 or at the next address in the program flow. All instructions in the program flow preceding this one will have completed execution and no subsequent instruction will have begun execution. This may be the address of the instruction that caused the interrupt or the next one (as in the case of a system call or trace interrupt). The SRR0 register is shown in [Figure 4-1](#).

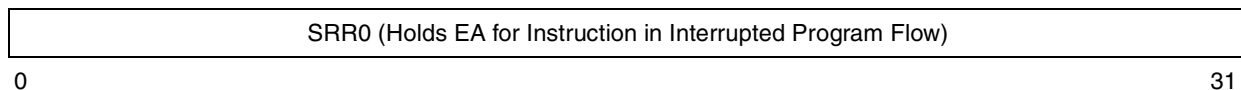


Figure 4-1. Machine Status Save/Restore Register 0 (SRR0)

SRR1 is used to save machine status (selected MSR bits and possibly other status bits) on interrupts and to restore those values when an **rfi** instruction is executed. SRR1 is shown in [Figure 4-2](#).

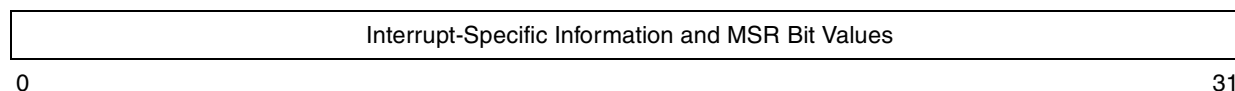


Figure 4-2. Machine Status Save/Restore Register 1 (SRR1)

Typically, when an interrupt occurs, SRR1[0–15] are loaded with interrupt-specific information and MSR[16–31] are placed into the corresponding bit positions of SRR1. For most interrupts, SRR1[0–5] and SRR1[7–15] are cleared, and MSR[6, 16–31] are placed into the corresponding bit positions of SRR1. [Table 4-4](#) provides a summary of the SRR1 bit settings when a machine check interrupt occurs. For a specific interrupt’s SRR1 bit settings, see [Section 4.6, “Interrupt Definitions.”](#)

The MSR is shown in [Figure 4-3](#).



Figure 4-3. Machine State Register (MSR)

The MSR bits are defined in [Table 4-4](#).

Table 4-4. MSR Bit Settings

Bits	Name	Description
0–5	—	Reserved
6	VEC ^{1, 2}	AltiVec vector unit available 0 The processor prevents dispatch of AltiVec instructions (excluding the data streaming instructions— dst , dstt , dstst , dststt , dss , and dssall). The processor also prevents access to the vector register file (VRF) and the vector status and control register (VSCR). Any attempt to execute an AltiVec instruction that accesses the VRF or VSCR, excluding the data streaming instructions generates the AltiVec unavailable interrupt. The data streaming instructions are not affected by this bit; the VRF and VSCR registers are available to the data streaming instructions even when the MSR[VEC] is cleared. 1 The processor can execute AltiVec instructions and the VRF and VSCR registers are accessible to all AltiVec instructions. Note that the VRSAVE register is not protected by MSR[VEC].
7–12	—	Reserved
13	POW ^{1, 3}	Power management enable 0 Power management disabled (normal operation mode). 1 Power management enabled (reduced power mode). Power management functions are implementation-dependent. See Chapter 9, “Power and Thermal Management.”
14	—	Reserved. Implementation-specific
15	ILE	Interrupt little-endian mode. When an interrupt occurs, this bit is copied into MSR[LE] to select the endian mode for the context established by the interrupt.

Table 4-4. MSR Bit Settings (continued)

Bits	Name	Description
16	EE	External interrupt enable 0 The core delays recognition of interrupts external to the core and decremter interrupt conditions. 1 The core is enabled to take an interrupt external to the core or the decremter interrupt.
17	PR ⁴	Privilege level 0 The processor can execute both user- and supervisor-level instructions. 1 The processor can only execute user-level instructions.
18	FP ²	Floating-point available 0 The core prevents dispatch of floating-point instructions, including floating-point loads, stores, and moves. 1 The core can execute floating-point instructions and can take floating-point enabled program interrupts.
19	ME	Machine check enable 0 Machine check interrupts are disabled. 1 Machine check interrupts are enabled.
20	FE0 ²	IEEE Std. 754 floating-point exception mode 0 (see Table 4-5)
21	SE	Single-step trace enable 0 The processor executes instructions normally. 1 The processor generates a single-step trace interrupt upon the successful execution of every instruction except rfi and sc . Successful execution means that the instruction caused no other interrupt.
22	BE	Branch trace enable 0 The processor executes branch instructions normally. 1 The processor generates a branch type trace interrupt when a branch instruction executes successfully.
23	FE1 ²	IEEE Std. 754 floating-point exception mode 1 (see Table 4-5)
24	—	Reserved. This bit corresponds to the AL bit of the POWER architecture.
25	IP	Interrupt prefix. The setting of this bit specifies whether an interrupt vector offset is prepended with Fs or Os. In the following description, <i>nnnn</i> is the offset of the interrupt. 0 Interrupts are vectored to the physical address 0x000 <i>n_nnnn</i> . 1 Interrupts are vectored to the physical address 0xFFFF <i>n_nnnn</i> .
26	IR ⁵	Instruction address translation 0 Instruction address translation is disabled. 1 Instruction address translation is enabled. For more information see Chapter 5, “Memory Management.”
27	DR ⁴	Data address translation 0 Data address translation is disabled. 1 Data address translation is enabled. For more information see Chapter 5, “Memory Management.”
28	—	Reserved

Table 4-4. MSR Bit Settings (continued)

Bits	Name	Description
29	PMM ¹	Performance monitor marked mode. e600-specific; defined as optional by the PowerPC architecture. PMM can be set when statistics need to be gathered on a specific (marked) process. Statistics are gathered only when the marked process is executing. For more information, see Section 10.4, “Event Counting.” 0 Process is not a marked process. 1 Process is a marked process.
30	RI	Indicates whether system reset or machine check interrupt is recoverable. The RI bit indicates whether from the perspective of the processor, it is safe to continue (that is, processor state data such as that saved to SRR0 is valid), but it does not guarantee that the interrupted process is recoverable. 0 Interrupt is not recoverable. 1 Interrupt is recoverable.
31	LE ⁶	Little-endian mode enable 0 The processor runs in big-endian mode. 1 The processor runs in little-endian mode.

¹ Optional to the PowerPC architecture

² A context synchronizing instruction must follow an mtmsr instruction.

³ A dssall and sync must precede an mtmsr instruction and then a context synchronizing instruction must follow.

⁴ A dssall and sync must precede an mtmsr and then a sync and context synchronizing instruction must follow. Note that if a user is not using the AltiVec data streaming instructions, then a dssall is not necessary prior to accessing the MSR[DR] or MSR[PR] bit.

⁵ A context synchronizing instruction must follow an mtmsr. When changing the MSR[IR] bit the context synchronizing instruction must reside at both the untranslated and the translated address following the mtmsr.

⁶ A dssall and sync must precede an rfi to guarantee a solid context boundary. Note that if a user is not using AltiVec data streaming instructions, a dssall is not necessary before accessing MSR[LE].

Note that setting MSR[EE] masks not only the architecture-defined external interrupt and decremter interrupts but also the e600-specific system management, and performance monitor interrupts.

The IEEE Std. 754 floating-point exception mode bits (FE0 and FE1) together define whether floating-point exceptions are handled precisely, imprecisely, or whether they are taken at all. As shown in [Table 4-5](#), if either FE0 or FE1 are set, the e600 core treats exceptions as precise. MSR bits are guaranteed to be written to SRR1 when the first instruction of the interrupt handler is encountered. For further details, see Chapter 2, “PowerPC Register Set” and Chapter 6, “Interrupts,” of *The Programming Environments Manual*.

Table 4-5. IEEE Std. 754 Floating-Point Exception Mode Bits

FE0	FE1	Mode
0	0	Floating-point exceptions disabled
0	1	Imprecise nonrecoverable. For this setting, the e600 operates in floating-point precise mode.
1	0	Imprecise recoverable. For this setting, the e600 operates in floating-point precise mode.
1	1	Floating-point precise mode

4.3.1 Enabling and Disabling Exceptions and Interrupts

When an exception condition exists that may generate an interrupt, it must be determined whether the interrupt is enabled for that condition as follows:

- System reset interrupts cannot be masked.
- A machine check interrupt can occur only if the machine check enable bit, MSR[ME], is set. If MSR[ME] is cleared, the processor goes directly into checkstop state when a machine check exception condition occurs. Individual machine check exceptions can be enabled and disabled through the following bits: HID1[EMCP], HID1[EBA], HID1[EBD], ICTRL[EIEC], ICTRL[EDCE], L2CR[L2PE], L2ERRINTEN[TPARINTEN], L2ERRINTEN[MBECCINTEN], and L2ERRINTEN[SBECCINTEN], which are described in [Table 4-8](#).
- Asynchronous, maskable interrupts (such as external and decremter interrupts) are enabled by setting MSR[EE]. If MSR[EE] = 0, recognition of these exception conditions is delayed. MSR[EE] is cleared automatically when an interrupt is taken to delay recognition of additional exceptions.
- The performance monitor interrupt is enabled for a specific process by setting MSR[PMM].
- The floating-point unavailable interrupt can be masked by setting MSR[FP].
- The AltiVec unavailable interrupt can be masked by setting MSR[VEC].
- IEEE Std. 754 floating-point enabled exceptions (a type of program exception) are ignored when both MSR[FE0] and MSR[FE1] are cleared. If either bit is set, all IEEE Std. 754 enabled floating-point exceptions can cause a program interrupt.
- The trace interrupt is enabled by setting either MSR[SE] or MSR[BE].
- Software table miss interrupts can be prevented by clearing HID0[STEN]. Note that this forces hardware table walks to be performed. See [Section 4.6.15, “TLB Miss Interrupts,”](#) for more information.

4.3.2 Steps for Interrupt Processing

After it is determined that the interrupt can be taken (all instruction-caused interrupts occurring earlier in the instruction stream have been handled, the instruction that caused the interrupt is next to be retired, and by confirming that the interrupt is enabled for the exception condition), the processor does the following:

1. SRR0 is loaded with an instruction address that depends on the type of interrupt. See the individual interrupt description for details about how this register is used for specific interrupts.
2. SRR1[0, 7–9] are cleared;
SRR1[1–5, 10–15] are loaded with information specific to the interrupt type;
and SRR1[6, 16–31] are loaded with a copy of the corresponding MSR bits.
3. The MSR is set as described in [Table 4-6](#). The new values take effect as the first instruction of the interrupt handler is fetched.

Note that MSR[IR] and MSR[DR] are cleared for all interrupt types; therefore, address translation is disabled for both instruction fetches and data accesses beginning with the first instruction of the interrupt handler.

4. Instruction fetch and execution resumes, using the new MSR value, at a location specific to the interrupt type. The location is determined by adding the interrupt's vector (see [Table 4-2](#)) to the base address determined by MSR[IP]. If IP is cleared, interrupts are vectored to the physical address 0x000n_nnnn. If IP is set, interrupts are vectored to the physical address 0xFFFFn_nnnn. For a machine check interrupt that occurs when MSR[ME] = 0 (machine check interrupts are disabled), the checkstop state is entered (the machine stops executing instructions). See [Section 4.6.2, “Machine Check Interrupt \(0x00200\).”](#)

4.3.3 Setting MSR[RI]

An operating system may handle MSR[RI] as follows:

- In the machine check and system reset interrupts—If MSR[RI] is cleared, the interrupt is not recoverable. If it is set, the interrupt is recoverable with respect to the processor.
- In each interrupt handler—When enough state information has been saved that a machine check or system reset interrupt can reconstruct the previous state, set MSR[RI].
- In each interrupt handler—Clear MSR[RI], set SRR0 and SRR1 appropriately, then execute **rfi**.
- Note that the RI bit being set indicates that, with respect to the processor, enough processor state data remains valid for the processor to continue, but it does not guarantee that the interrupted process can resume.

4.3.4 Returning from an Interrupt Handler

The Return from Interrupt (**rfi**) instruction performs context synchronization by allowing previously issued instructions to complete before returning to the interrupted process. In general, execution of the **rfi** instruction ensures the following:

- All previous instructions have completed to a point where they can no longer cause an interrupt.
- Previous instructions complete execution in the context (privilege, protection, and address translation) under which they were issued.
- The **rfi** instruction copies SRR1 bits back into the MSR.
- Instructions fetched after this instruction execute in the context established by this instruction.
- Program execution resumes at the instruction indicated by SRR0.

For a complete description of context synchronization, refer to Chapter 6, “Interrupts,” of *The Programming Environments Manual*.

4.4 Process Switching

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

- The **sync** instruction orders the effects of instruction execution. All instructions previously initiated appear to have completed before the **sync** instruction completes, and no subsequent instructions appear to be initiated until the **sync** instruction completes. For an example showing use of **sync**, see Chapter 2, “PowerPC Register Set,” of *The Programming Environments Manual*.
- The **isync** instruction waits for all previous instructions to complete and then discards any fetched instructions, causing subsequent instructions to be fetched (or refetched) from memory and to

execute in the context (privilege, translation, and protection) established by the previous instructions.

- The **stwcx.** instruction clears any outstanding reservations, ensuring that an **lwarx** instruction in an old process is not paired with an **stwcx.** instruction in a new one.

The operating system should set MSR[RI] as described in [Section 4.3.3, “Setting MSR\[RI\].”](#)

4.5 Data Stream Prefetching and Interrupts

As described in Chapter 5, “Cache, Interrupts, and Memory Management,” of the *AltiVec Technology Programming Environments Manual*, interrupts do not automatically cancel data stream prefetching. The operating system must stop streams explicitly when warranted—for example, when switching processes or changing virtual memory context. Care must be taken if data stream prefetching is used while in supervisor mode (MSR[PR] = 0).

4.6 Interrupt Definitions

[Table 4-6](#) shows all the types of interrupts that can occur with the e600 core and the MSR settings when the processor goes into supervisor mode due to an interrupt. Depending on the interrupt, certain of these bits are stored in SRR1 when an interrupt is taken.

Table 4-6. MSR Setting Due to Interrupt

Interrupt Type	MSR Bit Name/MSR Bit Number																
	VEC 6	POW 13	ILE 15	EE 16	PR 17	FP 18	ME 19	FE0 20	SE 21	BE 22	FE1 23	IP 25	IR 26	DR 27	PM 29	RI 30	LE 31
System reset	0	0	—	0	0	0	—	0	0	0	0	—	0	0	0	0	ILE
Machine check	0	0	—	0	0	0	0	0	0	0	0	—	0	0	0	0	ILE
DSI	0	0	—	0	0	0	—	0	0	0	0	—	0	0	0	0	ILE
ISI	0	0	—	0	0	0	—	0	0	0	0	—	0	0	0	0	ILE
External interrupt	0	0	—	0	0	0	—	0	0	0	0	—	0	0	0	0	ILE
Alignment	0	0	—	0	0	0	—	0	0	0	0	—	0	0	0	0	ILE
Program	0	0	—	0	0	0	—	0	0	0	0	—	0	0	0	0	ILE
Floating-point unavailable	0	0	—	0	0	0	—	0	0	0	0	—	0	0	0	0	ILE
Decrementer	0	0	—	0	0	0	—	0	0	0	0	—	0	0	0	0	ILE
System call	0	0	—	0	0	0	—	0	0	0	0	—	0	0	0	0	ILE
Trace	0	0	—	0	0	0	—	0	0	0	0	—	0	0	0	0	ILE
Performance monitor	0	0	—	0	0	0	—	0	0	0	0	—	0	0	0	0	ILE
AltiVec unavailable	0	0	—	0	0	0	—	0	0	0	0	—	0	0	0	0	ILE
ITLB miss	0	0	—	0	0	0	—	0	0	0	0	—	0	0	0	0	ILE
DTLB miss on load	0	0	—	0	0	0	—	0	0	0	0	—	0	0	0	0	ILE

Table 4-6. MSR Setting Due to Interrupt (continued)

Interrupt Type	MSR Bit Name/MSR Bit Number																
	VEC 6	POW 13	ILE 15	EE 16	PR 17	FP 18	ME 19	FE0 20	SE 21	BE 22	FE1 23	IP 25	IR 26	DR 27	PM 29	RI 30	LE 31
DTLB miss on store	0	0	—	0	0	0	—	0	0	0	0	—	0	0	0	0	ILE
Instruction address breakpoint	0	0	—	0	0	0	—	0	0	0	0	—	0	0	0	0	ILE
System management	0	0	—	0	0	0	—	0	0	0	0	—	0	0	0	0	ILE
AltiVec assist	0	0	—	0	0	0	—	0	0	0	0	—	0	0	0	0	ILE

Key: 0 Bit is cleared
 ILE Bit is copied from the MSR[ILE]
 — Bit is not altered
 Reserved bits are read as if written as 0

The setting of the interrupt prefix bit (IP) determines how interrupts are vectored. If IP is cleared, interrupts are vectored to the physical address $0x000n_nnnn$ (where n_nnnn is the vector offset); if IP is set, they are vectored to $0xFFFFn_nnnn$. Table 4-2 shows the interrupt vector offset of the first instruction of the interrupt handler routine for each interrupt type.

4.6.1 System Reset Interrupt (0x00100)

The e600 core implements the system reset interrupt as defined in the PowerPC architecture (OEA). The system reset interrupt is a nonmaskable, asynchronous interrupt signaled to the processor through the assertion of system-defined signals. In the e600 core, the interrupt is signaled by the assertion of either the \overline{hreset} or \overline{sreset} input signals, described more fully in Chapter 8, “Core Interface.”

A hard reset is initiated by asserting \overline{hreset} . A hard reset is used primarily for power-on reset (POR) (in which case \overline{trst} must also be asserted), but can also be used to restart a running processor. The \overline{hreset} signal must be asserted during power up and must remain asserted for a period that allows the PLL to achieve lock and the internal logic to be reset. This period is specified in the hardware specifications. If \overline{hreset} is asserted for less than the required interval, the results are not predictable.

If a hard reset request occurs (\overline{hreset} asserted), the processor immediately branches to the system reset interrupt vector ($0xFFFF0_0100$) without attempting to reach a recoverable state. If \overline{hreset} is asserted during normal operation, all operations cease and the machine state is lost. The e600 core internal state after a hard reset is defined in Table 2-36.

A soft reset is initiated by asserting \overline{sreset} . If \overline{sreset} is asserted, the processor is first put in a recoverable state. To do this, the e600 core allows any instruction at the point of completion to either complete or take an interrupt (note that load/store string or multiple accesses are not split), blocks completion of any following instructions and allows the completion queue to empty. If the soft reset request is made while the e600 core is in trace mode ($MSR[SE] = 1$ or $MSR[BE] = 1$), the interrupt is set as nonrecoverable and $SRR1[30]$ is cleared ($SRR1[30] = 0$). The state before the interrupt occurred is then saved as specified in the PowerPC architecture and instruction fetching begins at the system reset interrupt vector offset, $0x00100$. The vector base address for a soft reset depends on the setting of $MSR[IP]$ (either $0x0000_0100$ or $0xFFFF0_0100$). Soft resets are third in priority, after hard reset and machine check. Except for the trace

mode condition, this interrupt is recoverable provided attaining a recoverable state does not generate a machine check.

sreset is an edge-sensitive signal that can be asserted and negated asynchronously, provided there are two bus cycles in between. The system reset interrupt modifies the MSR, SRR0, and SRR1, as described in *The Programming Environments Manual*. Unlike hard reset, soft reset does not directly affect the states of output signals. Attempts to use *sreset* during a hard reset sequence or while the JTAG logic is non-idle can cause unpredictable results.

The e600 core implements HID0[NHR], which helps software distinguish a hard reset from a soft reset. Because this bit is cleared by a hard reset, but not by a soft reset, software can set this bit after a hard reset and determine whether a subsequent reset is a hard or soft reset (by examining whether this bit is still set). See Section 2.1.6.1, “Hardware Implementation-Dependent Register 0 (HID0).”

Table 4-7 lists register settings when a system reset interrupt is taken.

Table 4-7. System Reset Interrupt—Register Settings

Register	Setting Description			
SRR0	Cleared to zero by a hard reset On a soft reset, set to the effective address of the instruction that the processor would have attempted to execute next if no interrupt conditions were present.			
SRR1	0–5 Cleared 6 Loaded with equivalent MSR bit 7–15 Cleared 16–31 Loaded with equivalent MSR bits Note that if the processor state is corrupted to the extent that execution cannot resume reliably, MSR[RI] (SRR1[30]) is cleared.			
MSR	VEC 0 POW 0 ILE — EE 0 LE ILE	PR 0 FP 0 ME — FE0 0	SE 0 BE 0 FE1 0 IP —	IR 0 DR 0 PM 0 RI 0

Key: 0 Bit is cleared
 ILE Bit is copied from the MSR[ILE]
 — Bit is not altered

4.6.2 Machine Check Interrupt (0x00200)

The e600 core implements the machine check interrupt as defined in the PowerPC architecture (OEA). The e600 core conditionally initiates a machine check interrupt if MSR[ME] = 1 and an MPX bus error (*tea* assertion on data bus), assertion of the machine check (*mcp*) signal, address bus parity error on the MPX bus, data bus parity error on the MPX bus, L1 data cache error, L1 instruction cache error, or a core memory subsystem error is detected including:

- L2 data parity error
- L2 tag parity error
- Single-bit and multiple-bit L2 ECC errors

As defined in the PowerPC architecture, the interrupt is not taken if MSR[ME] is cleared, in which case the processor enters a checkstop state.

Certain machine check conditions can be enabled and disabled using HID1, ICTRL, and L2CR bits, as described in [Table 4-8](#).

Table 4-8. Machine Check Enable Bits

Bits	Name	Function
HID1[0]	EMCP	Enable \overline{mcp} . The primary purpose of this bit is to mask further machine check interrupts caused by assertion of \overline{mcp} , similar to how MSR[EE] can mask interrupts external to the core. 0 Masks \overline{mcp} . Assertion of \overline{mcp} does not generate a machine check interrupt or a checkstop. 1 Assertion of \overline{mcp} causes a checkstop if MSR[ME] = 0 or a machine check interrupt if MSR[ME] = 1.
HID1[2]	EBA	Enable/disable MPX bus address parity checking. 0 Prevents address parity checking. 1 Allows an address parity error to cause a checkstop if MSR[ME] = 0 or a machine check interrupt if MSR[ME] = 1. EBA and EBD allow the processor to operate with memory subsystems that do not generate parity.
HID1[3]	EBD	Enable MPX bus data parity checking 0 Parity checking is disabled. 1 Allows a data parity error to cause a checkstop if MSR[ME] = 0 or a machine check interrupt if MSR[ME] = 1. EBA and EBD allow the processor to operate with memory subsystems that do not generate parity.
ICTRL[4]	EIEC	Instruction cache parity error enable 0 When the bit is cleared, any parity error in the L1 instruction cache is masked and does not cause machine checks or checkstop 1 Enables instruction cache parity errors. When an instruction cache parity error occurs, a machine check interrupt is taken if MSR[ME] = 1. When this condition occurs, SRR1[1] is set. For details, see Section 4.6.2, “Machine Check Interrupt (0x00200).”
ICTRL[5]	EDCE	Data cache parity error enable 0 When the bit is cleared, any parity error in the L1 data cache is masked and does not cause machine checks or checkstop 1 Enables data cache parity errors. When a data cache parity error occurs, a machine check interrupt is taken if MSR[ME] = 1. When this condition occurs, SRR1[2] is set. For details, see Section 4.6.2, “Machine Check Interrupt (0x00200).”
L2CR[1]	L2PE	L2 data parity checking enable 0 L2 data parity checking disabled 1 L2 data parity checking enabled if L2ERRDIS[MBECCDIS]=1 and L2ERRDIS[SBECCDIS]=1. If ECC is enabled (L2ERRDIS[MBECCDIS]=0 or L2ERRDIS[SBECCDIS]=0), setting L2PE has no effect; ECC checking will still be performed. Note: The L2ERRDIS register includes bits to enable/disable tag parity checking. Data parity can only be enabled with L2CR[L2PE] if ECC is disabled in the L2ERRDIS register. By default, tag parity and data ECC checking are enabled on the e600 core.
L2ERRINTEN[28] ¹	MBECCINTEN	Multiple-bit ECC error reporting enable 0 Multiple-bit ECC error reporting disabled 1 Multiple-bit ECC error reporting enabled

Table 4-8. Machine Check Enable Bits (continued)

Bits	Name	Function
L2ERRINTEN[29] ¹	SBECCINTEN	Single-bit ECC error reporting enable 0 Single-bit ECC error reporting disabled 1 Single-bit ECC error reporting enabled
L2ERRINTEN[27] ¹	TPARINTEN	Tag parity error reporting enable 0 Tag parity error reporting disabled 1 Tag parity error reporting enabled.

¹ The corresponding bit in L2ERRDIS must be cleared (enabled).

A \overline{tea} indication on the MPX bus can result from any load or store operation initiated by the processor. In general, \overline{tea} is expected to be used by a memory controller to indicate that a memory parity error or an uncorrectable memory ECC error has occurred. Note that the resulting machine check interrupt is imprecise and unordered with respect to the instruction that originated the MPX bus operation.

For other core memory subsystem errors, if MSR[ME] and the appropriate HID1, ICTRL, and L2CR bits are set, the interrupt is recognized and handled; otherwise, in most cases, the processor generates an internal checkstop condition (an example of an exception to this rule is if MSR[ME] = 1, HID1[EMCP] = 0, and \overline{mcp} is asserted, then \overline{mcp} is ignored and neither a machine check interrupt nor checkstop occur). When a processor is in checkstop state, instruction processing is suspended and generally cannot continue without restarting the processor. Note that many conditions may lead to the checkstop condition; the disabled machine check interrupt is only one of these.

A machine check interrupt may result from referencing a nonexistent physical address, either directly (with MSR[DR] = 0) or through an invalid translation. If a **dcbz** instruction introduces a block into the cache associated with a nonexistent physical address, a machine check interrupt can be delayed until an attempt is made to store that block to main memory. Not all processors that implement the PowerPC architecture provide the same level of error checking. Checkstop sources are implementation-dependent.

Machine check interrupts are enabled when MSR[ME] = 1; this is described in [Section 4.6.2.1, “Machine Check Interrupt Enabled \(MSR\[ME\] = 1\).”](#) If MSR[ME] = 0 and a machine check occurs, the processor enters the checkstop state. The checkstop state is described in [Section 4.6.2.2, “Checkstop State \(MSR\[ME\] = 0\).”](#)

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

Machine check interrupts are enabled when MSR[ME] = 1. When a machine check condition occurs, the e600 core waits for the processor to quiesce (defined in the “Glossary of Terms and Abbreviations”) and the core memory subsystem to empty all its queues and terminate all pending data tenures. Then the vector touch engine (VTE) stops all streams when a machine check is detected. Once the processor and the core memory subsystem have quiesced, a machine check interrupt is taken. When a machine check interrupt is taken, registers are updated as shown in [Table 4-9](#).

Table 4-9. Machine Check Interrupt—Register Settings

Register	Setting Description			
SRR0	On a best-effort basis the e600 core sets this to an EA of some instruction that was executing or about to be executing when the machine check condition occurred.			
SRR1	0 Cleared 1 L1 instruction cache error 2 L1 data cache error 3–5 Normally cleared, used in debug. 6 Loaded with equivalent MSR bit 7–9 Cleared 10 Normally cleared, used in debug. 11 MSS error. Set for an L2 cache tag parity or L2 data parity error. Refer to Section 2.1.6.4, “Memory Subsystem Status Register (MSSSR0),” for more information. 12 MCP. Set when <i>mcp</i> signal is asserted; otherwise 0 13 TEA. Set when <i>tea</i> signal is asserted; otherwise 0 14 DP. Set when a data bus parity error is detected on the MPX bus; otherwise 0 15 AP. Set when an address bus parity error is detected on the MPX bus; otherwise 0 16–29 Loaded with equivalent MSR bits 30 Set in case of a recoverable interrupt 31 Loaded with equivalent MSR bits			
L2ERRDET	0 Multiple L2 errors (Bit reset, write-1-to-clear) 1–26 Reserved 27 Tag parity error (Bit reset, write-1-to-clear) 28 Multiple-bit ECC error (Bit reset, write-1-to-clear) 29 Single-bit ECC error (Bit reset, write-1-to-clear) 30–31 Reserved			
MSSSR0	0–12 Normally cleared, used in debug, writing nonzero values may cause boundedly undefined results 13 L2 tag parity error 14 L2 data parity error 15 Normally cleared, used in debug, writing nonzero values may cause boundedly undefined results 16 Normally cleared, used in debug, writing nonzero values may cause boundedly undefined results. 17 Address bus parity error 18 Data bus parity error 19 Bus transfer error acknowledge 20–31 Reserved			
MSR	VEC 0 POW 0 ILE — EE 0 PR 0	FP 0 ME 0 FE0 0 SE 0	BE 0 FE1 0 IP — IR 0	DR 0 PM 0 RI 0 LE ILE

Key: 0 Bit is cleared
 ILE Bit is copied from the MSR[ILE]
 — Bit is not altered

Note that to handle another machine check interrupt, the handler should set MSR[ME] as soon as it is practical after a machine check interrupt is taken. Otherwise, subsequent machine check interrupts cause the processor to enter checkstop state.

When the e600 core takes the machine check interrupt, it sets one or more error bits in SRR1. The e600 has multiple data parity error sources that can cause a machine check interrupt. The MSS error indicates one of many possible L2 parity errors as described more completely in [Section 2.1.6.4, “Memory Subsystem Status Register \(MSSSR0\).”](#) Memory subsystem errors in an ICTRL field need to be enabled

to cause an error, see [Section 2.1.6.5, “Instruction and Data Cache Registers,”](#) for details. The SRR1[MCP] bit (SRR1[12]) indicates that the machine check signal was asserted. The TEA bit (SRR1[13]) indicates that the machine check was caused by a \overline{tea} assertion on the MPX bus.

The machine check interrupt is usually unrecoverable in the sense that execution cannot resume in the context that existed before the interrupt. If the condition that caused the machine check does not otherwise prevent continued execution, MSR[ME] is set by software to allow the processor to continue execution at the machine check interrupt vector address. Typically, earlier processes cannot resume; however, operating systems can use the machine check interrupt handler to try to identify and log the cause of the machine check condition.

When a machine check interrupt is taken, instruction fetching resumes at offset 0x00200 from the physical base address indicated by MSR[IP].

4.6.2.2 Checkstop State (MSR[ME] = 0)

If MSR[ME] = 0 and a machine check condition occurs, the processor enters the checkstop state.

When a processor is in checkstop state, instruction processing is suspended and generally cannot resume without the processor being reset. The contents of all latches are frozen within six cycles upon entering checkstop state.

Note that the e600 core has a $\overline{ckstp_out}$ signal (open-drain) that is asserted when the e600 enters the checkstop state. Also, logic external to the core can cause the core to enter the checkstop state by asserting $\overline{ckstp_in}$. See [Chapter 8, “Core Interface,”](#) for more information on these checkstop signals.

4.6.3 DSI Interrupt (0x00300)

A DSI interrupt occurs when no higher priority interrupt exists and an error condition related to a data memory access occurs. The DSI interrupt is implemented as it is defined in the PowerPC architecture (OEA). For details on the DSI interrupt, see “DSI Interrupt (0x00300),” in *The Programming Environments Manual*. For example, a **lwarx** or **stwcx** instruction that addresses memory to be mapped with the write-through (W = 1) or caching-inhibited (I = 1) attribute causes a DSI interrupt.

4.6.3.1 DSI Interrupt—Page Fault

When hardware table searching is enabled, HID0[STEN] = 0, and there is a TLB miss for a load, store, or cache operation, a DSI interrupt is taken if the resulting hardware table search causes a page fault. When software table searching is enabled, HID0[STEN] = 1, the TLB miss handlers configure SRR1 and DSISR appropriately for a page fault in this case and branch to the DSI interrupt handlers as described in [Section 5.5.5.2, “Example Software Table Search Operation.”](#)

The condition that caused the interrupt is defined in the DSISR. These conditions also use the data address register (DAR) as shown in [Table 4-10](#).

Table 4-10. DSI Interrupt—Register Settings

Register	Setting Description
DSISR	0 Cleared
	1 Set by the hardware (if HID0[STEN]=0) or the DTLB miss interrupt handler if the translation of an attempted access is not found in the primary page table entry group (PTEG), or in the rehashed secondary PTEG, or in the range of a DBAT register; otherwise cleared.
	2–3 Cleared
	4 Set if a memory access is not permitted by the page or BAT protection mechanism; otherwise cleared.
	5 Set if the lwarx or stwcx . instruction is attempted to write-through (W = 1) or caching-inhibited (I = 1) memory.
	6 Set for a store operation and cleared for a load operation.
	7–8 Cleared
	9 Set if DABR match occurs, otherwise cleared.
	10 Cleared
	11 Set if eciwx or ecowx instruction is executed when EAR[E] = 0; otherwise cleared.
	12–31 Cleared
	DAR

4.6.3.2 DSI Interrupt—Data Address Breakpoint Facility

The e600 core also implements the data address breakpoint facility, which is defined as optional in the PowerPC architecture and is supported by the optional data address breakpoint register (DABR) and the DSI interrupt. Although the architecture does not strictly prescribe how this facility must be implemented, the e600 core follows the recommendations provided by the architecture and described in Chapter 2, “Programming Model,” and Chapter 6 “Interrupts,” in *The Programming Environments Manual*. The granularity of the data address breakpoint compare is a double word for all accesses except AltiVec quad-word loads and stores. For AltiVec accesses, the least significant bit of the DAB field (DABR[28]) is ignored, thus providing quad-word granularity. For these quad-word DAB matches, the DAR register is loaded with a quad-word-aligned address.

When a DSI interrupt is taken, instruction fetching resumes at offset 0x00300 from the physical base address indicated by MSR[IP].

4.6.4 ISI Interrupt (0x00400)

An ISI interrupt occurs when no higher priority interrupt exists and an attempt to fetch the next instruction fails. This interrupt is implemented as it is defined by the PowerPC architecture (OEA), and is taken for the following conditions:

- The effective address cannot be translated.
- The fetch access is to a no-execute segment (SR[N] = 1).
- The fetch access is to guarded storage and MSR[IR] = 1.
- The fetch access violates memory protection.

When an ISI interrupt is taken, instruction fetching resumes at offset 0x00400 from the physical base address indicated by MSR[IP].

4.6.5 External Interrupt (0x00500)

An interrupt external to the core is signaled to the core by the assertion of the external interrupt signal (\overline{int}) when $MSR[EE] = 1$. The \overline{int} signal is expected to remain asserted until the e600 core takes the external interrupt. If \overline{int} is negated early, recognition of the interrupt request is not guaranteed. After the e600 core begins execution of the external interrupt handler, the system can safely negate \overline{int} . When the core detects assertion of \overline{int} , it stops dispatching and waits for all pending instructions to complete, including string and multiple instructions. This allows any instructions in progress that need to take an interrupt to do so before the external interrupt is taken. After all instructions have vacated the completion buffer, the e600 core takes the external interrupt as defined in the PowerPC architecture (OEA).

The e600 core also allows supervisor software to cause an external interrupt through the $ICTRL[CIRQ]$ bit. When $ICTRL[CIRQ]$ is set (and $MSR[EE] = 1$), the core functions as if \overline{int} has been asserted, and it stop dispatching and waits for all pending instructions to complete. After all instructions have vacated the completion buffer, the core takes the external interrupt. Note that if both $ICTRL[CIRQ]$ is set and \overline{int} is asserted, only one interrupt is taken. Refer to [Section 2.1.6.5.15, “Instruction Cache and Interrupt Control Register \(ICTRL\),”](#) for more information on the setting and clearing of the $ICTRL[CIRQ]$ bit.

An external interrupt may be delayed by other higher priority interrupts or if $MSR[EE]$ is cleared when the interrupt occurs.

When an external interrupt is taken, instruction fetching resumes at offset 0x00500 from the physical base address indicated by $MSR[IP]$.

Table 4-11 lists register settings when an external interrupt is taken.

Table 4-11. External Interrupt—Register Settings

Register	Setting Description			
SRR0	Set to the effective address of the instruction that the processor would have attempted to execute next if no interrupt conditions were present.			
SRR1	0 Cleared 1 Set when an external interrupt is caused by the $ICTRL[CIRQ]$ bit 2–5 Cleared 6 Loaded with equivalent MSR bits 7–9 Cleared 10 Set when an external interrupt is caused by \overline{int} assertion 11–15 Cleared 16–31 Loaded with equivalent MSR bits			
MSR	VEC 0 POW 0 ILE — EE 0 LE ILE	PR 0 FP 0 ME — FE0 0	SE 0 BE 0 FE1 0 IP —	IR 0 DR 0 PM 0 RI 0

Key: 0 Bit is cleared
 ILE Bit is copied from the $MSR[ILE]$
 — Bit is not altered

4.6.6 Alignment Interrupt (0x00600)

The e600 core implements the alignment interrupt as defined by the PowerPC architecture (OEA). An alignment interrupt is initiated when any of the following occurs:

- The operand of a floating-point load or store is not word-aligned.
- The operand of **lmw**, **stmw**, **lwarx**, or **stwcx** is not word-aligned.
- The operand of **dcbz** is in a page that is write-through or cache-inhibited.
- An attempt is made to execute **dcbz** when the data cache is disabled or locked.
- A multiple or string access is attempted with MSR[LE] set

Note that the e600 core does not take an alignment interrupt for load/store string accesses that cross a protection boundary or for a load/store multiplex access that crosses a segment or BAT boundary.

When an alignment interrupt is taken, instruction fetching resumes at offset 0x00600 from the physical base address indicated by MSR[IP].

The register settings for alignment interrupts are shown in [Table 4-12](#).

Table 4-12. Alignment Interrupt—Register Settings

Register	Setting
DSISR	0—14 Cleared 15—16 For instructions that use register indirect with index addressing—set to bits 29–30 of the instruction. For instructions that use register indirect with immediate index addressing—cleared. 17 For instructions that use register indirect with index addressing—set to bit 25 of the instruction. For instructions that use register indirect with immediate index addressing— Set to bit 5 of the instruction 18—21 For instructions that use register indirect with index addressing—set to bits 21–24 of the instruction. For instructions that use register indirect with immediate index addressing—set to bits 1–4 of the instruction. 22–26 Set to bits 6–10 (identifying either the source or destination) of the instruction. Undefined for dcbz . 27–31 Set to bits 11–15 of the instruction (rA) for instructions that use the update form. For lmw , lswi , and lswx instructions, set to either bits 11–15 of the instruction or to any register number not in the range of registers loaded by a valid form instruction. Otherwise undefined.
DAR	Set to the EA of the data access as computed by the instruction causing the alignment interrupt.

4.6.7 Program Interrupt (0x00700)

The e600 core implements the program interrupt as it is defined by the PowerPC architecture (OEA). A program interrupt occurs when no higher priority interrupt exists and one or more of the interrupt conditions defined in the OEA occur.

The e600 core invokes the system illegal instruction program interrupt when it detects any instruction from the illegal instruction class. The core fully decodes the SPR field of the instruction. If an undefined SPR is specified, a program interrupt is taken.

The UISA defines **mtspr** and **mfspr** with the record bit (Rc) set as causing a program interrupt or giving a boundedly undefined result. In the e600 core, the appropriate condition register (CR) should be treated as undefined. Likewise, the PowerPC architecture states that the Floating Compared Unordered (**fcmpu**) or Floating Compared Ordered (**fcmpo**) instructions with the record bit set can either cause a program interrupt or provide a boundedly undefined result. In the e600 core, the BF field in an instruction encoding for these cases is considered undefined.

The e600 core does not support either of the two floating-point imprecise modes supported by the PowerPC architecture. Unless interrupts are disabled ($MSR[FE0] = MSR[FE1] = 0$), all floating-point interrupts are treated as precise.

When a program interrupt is taken, instruction fetching resumes at offset 0x00700 from the physical base address indicated by MSR[IP]. Chapter 6, “Interrupts,” in *The Programming Environments Manual* describes register settings for this interrupt.

4.6.8 Floating-Point Unavailable Interrupt (0x00800)

The floating-point unavailable interrupt is implemented as defined in the PowerPC architecture. A floating-point unavailable interrupt occurs when no higher priority interrupt exists, an attempt is made to execute a floating-point instruction (including floating-point load, store, or move instructions), and the floating-point available bit in the MSR is disabled, ($MSR[FP] = 0$). Register settings for this interrupt are described in Chapter 6, “Interrupts,” in *The Programming Environments Manual*.

When a floating-point unavailable interrupt is taken, instruction fetching resumes at offset 0x00800 from the physical base address indicated by MSR[IP].

4.6.9 Decrementer Interrupt (0x00900)

The decrementer interrupt is implemented in the e600 core as it is defined by the PowerPC architecture. The decrementer interrupt occurs when no higher priority interrupt exists, a decrementer interrupt condition occurs (for example, the decrementer register has completed decrementing), and $MSR[EE] = 1$. In the e600 core, the decrementer register is decremented at one fourth the bus clock rate. Register settings for this interrupt are described in Chapter 6, “Interrupts,” in *The Programming Environments Manual*.

When a decrementer interrupt is taken, instruction fetching resumes at offset 0x00900 from the physical base address indicated by MSR[IP].

4.6.10 System Call Interrupt (0x00C00)

A system call interrupt occurs when a System Call (**sc**) instruction is executed. In the e600 core, the system call interrupt is implemented as it is defined in the PowerPC architecture. Register settings for this interrupt are described in Chapter 6, “Interrupts,” in *The Programming Environments Manual*.

When a system call interrupt is taken, instruction fetching resumes at offset 0x00C00 from the physical base address indicated by MSR[IP].

4.6.11 Trace Interrupt (0x00D00)

The trace interrupt is taken if $MSR[SE] = 1$ or if $MSR[BE] = 1$ and the currently completing instruction is a branch. Each instruction considered during trace mode completes before a trace interrupt is taken. When a **mtmsr** instruction is executed and the MSR[SE] transitions from 0 to 1, following the completion of that **mtmsr**, a trace interrupt is taken.

When a trace interrupt is taken, instruction fetching resumes at offset 0x00D00 from the base address indicated by MSR[IP].

4.6.12 Floating-Point Assist Interrupt (0x00E00)

The optional floating-point assist interrupt defined by the PowerPC architecture is not implemented in the e600 core.

4.6.13 Performance Monitor Interrupt (0x00F00)

The e600 core provides a performance monitor facility to monitor and count predefined events such as core clocks, misses in either the instruction cache or the data cache, instructions dispatched to a particular execution unit, mispredicted branches, and other occurrences. An overflow of the counter in such events can be used to trigger the performance monitor interrupt. The performance monitor facility is not defined by the PowerPC architecture.

The performance monitor provides the ability to generate a performance monitor interrupt triggered by an enabled condition or event. This interrupt is triggered by an enabled condition or event defined as follows:

- A PMC_x register overflow condition occurs
 - $MMCR0[PMC1CE]$ and $PMC1[OV]$ are both set
 - $MMCR0[PMC_nCE]$ and $PMC_n[OV]$ are both set ($n > 1$)
- A time base event— $MMCR0[TBEE] = 1$ and the TBL bit specified in $MMCR0[TBSEL]$ changes from 0 to 1

$MMCR0[PMXE]$ must be set for any of these conditions to signal a performance monitor interrupt.

Although the performance monitor interrupt may occur with $MSR[EE] = 0$, the interrupt is not taken until $MSR[EE] = 1$.

As a result of a performance monitor interrupt being generated, the performance monitor saves in the SIAR the effective address of the last instruction completed before the interrupt is generated. Note that SIAR is not updated if performance monitor counting has been disabled by setting $MMCR0[0]$.

The performance monitor can be used for the following:

- To increase system performance with efficient software, especially in a multiprocessing system. Memory hierarchy behavior must be monitored and studied to develop algorithms that schedule tasks (and perhaps partition them) and that structure and distribute data optimally.
- To help system developers bring up and debug their systems.

The performance monitor uses the following SPRs:

- The performance monitor counter registers ($PMC1$ – $PMC6$) are used to record the number of times a certain event has occurred. $UPMC1$ – $UPMC6$ provide user-level read access to these registers.
- The monitor mode control registers ($MMCR0$ – $MMCR2$) are used to enable various performance monitor interrupt functions. $UMMCR0$ – $UMMCR2$ provide user-level read access to these registers.
- The sampled instruction address register (SIAR) contains the effective address of an instruction executing at or around the time that the processor signals the performance monitor interrupt condition. The USIAR provides user-level read access to the SIAR.

Table 4-13 lists register settings when a performance monitor interrupt is taken.

Table 4-13. Performance Monitor Interrupt—Register Settings

Register	Setting Description			
SRR0	Set to the effective address of the instruction that the processor would have attempted to execute next if no interrupt conditions were present.			
SRR1	0–5 Cleared 6 Loaded with equivalent MSR bit 7–15 Cleared 16–31 Loaded with equivalent MSR bits			
MSR	VEC 0 POW 0 ILE — EE 0 LE ILE	PR 0 FP 0 ME — FE0 0	SE 0 BE 0 FE1 0 IP —	IR 0 DR 0 PM 0 RI 0

Key: 0 Bit is cleared
 ILE Bit is copied from the MSR[ILE]
 — Bit is not altered

The performance monitor interrupt follows the normal PowerPC interrupt model with a defined interrupt vector offset (0x00F00). The priority of the performance monitor interrupt lies between the external interrupt and the decremter interrupt (see Table 4-3). The contents of the SIAR are described in Section 2.1.6.9, “Performance Monitor Registers.” The performance monitor is described in Chapter 10, “Performance Monitor.”

4.6.14 AltiVec Unavailable Interrupt (0x00F20)

The AltiVec facility includes another instruction-caused, precise interrupt in addition to the interrupts defined by the PowerPC architecture (OEA). An AltiVec unavailable interrupt occurs when no higher priority interrupt exists (see Table 4-3), and an attempt is made to execute an AltiVec instruction that accesses the vector register (VR) or the vector status and control register (VSCR) when MSR[VEC] = 0.

Note that the data streaming instructions, **dss**, **dst**, and **dstst** do not cause an AltiVec unavailable interrupt: the VR and VSCR registers are available to the data streaming instructions even when MSR[VEC] = 0.

4.6.15 TLB Miss Interrupts

When software table searching is enabled (HID0[STEN] = 1), and the effective address for a fetch can not be translated by the BATs or core TLBs, one of three TLB miss interrupts is generated:

- ITLB miss interrupt
- DTLB miss-on-load
- DTLB miss-on-store

When the interrupt occurs, the effective address of the access that requires the software table search is saved in the TLBMISS register. Also, when the interrupt occurs, the fields of the PTEHI register are loaded automatically with the corresponding SR[VSID] information and the API of the missed page address.

These registers are set to facilitate the searching of the page tables in software and their settings are shown in this section.

As described in the example code ([Section 5.5.5.2.2, “Code for Example Interrupt Handlers”](#)), if a TLB miss interrupt handler fails to find the desired PTE, then a page fault must be synthesized.

An example code sequence for a software table search operation (including a handler for these interrupts) is provided in [Section 5.5.5.2, “Example Software Table Search Operation.”](#)

[Table 4-14](#) details the register settings when one of the TLB miss interrupts occurs.

Table 4-14. TLB Miss Interrupts—Register Settings

Register	Setting Description			
TLBMISS	0–30	Effective page address for the access that caused the TLB miss interrupt		
	31	LRU Way		
PTEHI	0	Set to 1		
	1–24	The virtual segment ID (VSID) of the missed page address, SR[VSID] is copied to this field.		
	25	Set to 0		
	26–31	The effective address’s abbreviated page index (EA[API]).		
SRR0	Set to the effective address of the instruction that the processor would have attempted to execute next if no interrupt conditions were present.			
SRR1	0–5	Cleared		
	6	Loaded with equivalent MSR bits		
	7–11	Cleared except when DTLB miss on store interrupt occurs with C = 0, then SRR1[11] = 1. Refer to Section 4.6.15.3, “Data Table Miss-on-Store Interrupt—DTLB Miss-on-Store (0x01200)” , for details.		
	12	Key for TLB Miss When the access is a user access (MSR[PR] = 0), this bit is set equal to SR[Ks]. When access is a supervisor access (MSR[PR] = 1), this bit is set equal to SR[Kp].		
	13–15	Cleared		
	16–31	Loaded with equivalent MSR bits		
MSR	VEC	0	PR	0
	POW	0	FP	0
	ILE	—	ME	—
	EE	0	FE0	0
	LE	ILE	SE	0
			BE	0
		FE1	0	
		IP	—	
		IR	0	
		DR	0	
		PM	0	
		RI	0	

Key: 0 Bit is cleared
 ILE Bit is copied from the MSR[ILE]
 — Bit is not altered

4.6.15.1 Instruction Table Miss Interrupt—ITLB Miss (0x01000)

When software table searching is enabled (HID0[STEN] = 1), and the effective address for an instruction fetch cannot be translated by the IBATs or ITLB, an ITLB miss interrupt is generated. [Table 4-14](#) details the register settings for TLBMISS and PTEHI when an ITLB miss interrupt occurs

When an instruction TLB miss interrupt is taken, instruction execution for the handler begins at offset 0x01000 from the physical base address indicated by MSR[IP].

4.6.15.2 Data Table Miss-on-Load Interrupt—DTLB Miss-on-Load (0x01100)

When software table searching is enabled (HID0[STEN] = 1), and the effective address for a load or cache load operation cannot be translated by the DBATs or DTLB, a DTLB miss on load interrupt is generated. If a TLB miss occurs in the middle of a load string or multiple access, the e600 core takes the DTLB miss-on-load interrupt when it occurs; after the interrupt is handled, the instruction is restarted. [Table 4-14](#) details the register settings for the TLBMISS and PTEHI when a DTLB miss-on-load interrupt occurs.

When a DTLB miss on load interrupt is taken, instruction execution for the handler begins at offset 0x01100 from the physical base address indicated by MSR[IP].

4.6.15.3 Data Table Miss-on-Store Interrupt—DTLB Miss-on-Store (0x01200)

When the effective address for a data store or cache store operation can not be translated by the DBAT or DTLB, a DTLB miss-on-store interrupt is generated. The data TLB miss-on-store interrupt is also taken when the changed bit for a matching DTLB entry needs to be updated in memory for a store operation (C = 0). If a TLB miss occurs in the middle of a store string or multiple access, the e600 core takes the DTLB miss-on-store interrupt.

[Table 4-14](#) details the register settings for TLBMISS and PTEHI when a TLB miss interrupt occurs. Note that SRR1[11] is set when a DTLB hit occurs and the matching entry must have its changed bit in the PTE set due to a data store operation (PTE C bit = 0, and must be set to 1).

When a data TLB miss-on-store interrupt is taken, instruction execution for the handler begins at offset 0x01200 from the physical base address indicated by MSR[IP].

4.6.16 Instruction Address Breakpoint Interrupt (0x01300)

An instruction address breakpoint interrupt occurs when all of the following conditions are met:

- The instruction breakpoint address IABR[0–29] matches EA[0–29] of the next instruction to complete in program order. The instruction that triggers the instruction address breakpoint interrupt is not executed before the interrupt handler is invoked.
- The IABR[TE] bit matches the MSR[IR] bit.
- The breakpoint enable bit (IABR[BE]) is set.

The instruction tagged with the match does not complete before the breakpoint interrupt is taken.

[Table 4-15](#) lists register settings when an instruction address breakpoint interrupt is taken.

Table 4-15. Instruction Address Breakpoint Interrupt—Register Settings

Register	Setting Description
SRR0	Set to the effective address of the instruction that the processor would have attempted to execute next if no interrupt conditions were present.
SRR1	0–5 Cleared 6 Loaded with equivalent MSR bit 7–15 Cleared 16–31 Loaded with equivalent MSR bits

Table 4-15. Instruction Address Breakpoint Interrupt—Register Settings (continued)

MSR	VEC	0	PR	0	SE	0	IR	0			
	POW	0		FP		0		BE	0	DR	0
	ILE	—		ME		—		FE1	0	PM	0
	EE	0		FE0		0		IP	—	RI	0
	LE	Set to value of ILE									

Key: 0 Bit is cleared
 ILE Bit is copied from the MSR[ILE]
 — Bit is not altered

The e600 core requires that an **mtspr** to the IABR be followed by a context-synchronizing instruction. The core cannot generate a breakpoint response for that context-synchronizing instruction if the breakpoint is enabled by the **mtspr**[IABR] immediately preceding it. The core also cannot block a breakpoint response on the context-synchronizing instruction if the breakpoint was disabled by the **mtspr**[IABR] instruction immediately preceding it. The format of the IABR register is shown in [Section 2.1.6.6, “Instruction Address Breakpoint Register \(IABR\).”](#)

When an instruction address breakpoint interrupt is taken, instruction fetching resumes at offset 0x01300 from the base address indicated by MSR[IP].

4.6.17 System Management Interrupt (0x01400)

The e600 core implements a system management interrupt, which is not defined by the PowerPC architecture. The system management interrupt is very similar to the external interrupt and it must be enabled with MSR[EE] = 1. It is particularly useful in implementing the nap mode. It has priority over an external interrupt (see [Table 4-3](#)) and uses a different vector in the interrupt table (offset 0x01400).

[Table 4-16](#) lists register settings when a system management interrupt is taken.

Table 4-16. System Management Interrupt—Register Settings

Register	Setting Description										
SRR0	Set to the effective address of the instruction that the processor would have attempted to execute next if no interrupt conditions were present.										
SRR1	0–5 Cleared										
	6 Loaded with equivalent MSR bit										
	7–15 Cleared										
	16–31 Loaded with equivalent MSR bits										
MSR	VEC	0	PR	0	SE	0	IR	0			
	POW	0		FP		0		BE	0	DR	0
	ILE	—		ME		—		FE1	0	PM	0
	EE	0		FE0		0		IP	—	RI	0
	LE	Set to value of ILE									

Key: 0 Bit is cleared
 ILE Bit is copied from the MSR[ILE]
 — Bit is not altered

Like the external interrupt, a system management interrupt is signaled to the e600 core by the assertion of an input signal. The system management interrupt signal (\overline{smi}) is expected to remain asserted until the

interrupt is taken. If \overline{smi} is negated early, recognition of the interrupt request is not guaranteed. After the e600 core begins execution of the system management interrupt handler, the system can safely negate \overline{smi} . After the assertion of \overline{smi} is detected, the e600 core stops dispatching instructions and waits for all pending instructions to complete. This allows any instructions in progress that need to take an interrupt to do so before the system management interrupt is taken. Note that the core waits for any load/store string or multiple instructions that have begun to be complete before taking the system management interrupt.

When a system management interrupt is taken, instruction fetching resumes as offset 0x01400 from the base address indicated by MSR[IP].

4.6.18 AltiVec Assist Interrupt (0x01600)

The e600 core implements an AltiVec assist interrupt to handle denormalized numbers in Java mode (VSCR[NJ] = 0). An AltiVec assist interrupt occurs when no higher priority interrupt exists and an instruction causes a trap condition, as defined in Section 7.1.2.5, “Java Mode, NaNs, Denormalized Numbers, and Zeros.” Note that the core handles most denormalized numbers in Java mode by trapping to the AltiVec assist interrupt, but for some instructions, the core produces the exact result without trapping.

Table 4-16 lists register settings when an AltiVec assist interrupt is taken.

Table 4-17. AltiVec Assist Interrupt—Register Settings

Register	Setting Description			
SRR0	Set to the effective address of the instruction that caused the interrupt.			
SRR1	0–5 Cleared 6 Loaded with equivalent MSR bit 7–15 Cleared 16–31 Loaded with equivalent MSR bits			
MSR	VEC 0 POW 0 ILE — EE 0 LE Set to value of ILE	PR 0 FP 0 ME — FE0 0	SE 0 BE 0 FE1 0 IP —	IR 0 DR 0 PM 0 RI 0

Key: 0 Bit is cleared
 ILE Bit is copied from the MSR[ILE]
 — Bit is not altered

When an AltiVec assist interrupt is taken, instruction fetching resumes at offset 0x01600 from the base address indicated by MSR[IP].

Chapter 5

Memory Management

This chapter describes the e600 core's implementation of the memory management unit (MMU) specifications provided by the operating environment architecture (OEA) for processors that implement the PowerPC architecture. The primary function of the MMU in a processor is the translation of logical (effective) addresses to physical addresses (referred to as real addresses in the architecture specification) for memory accesses and I/O accesses (I/O accesses are assumed to be memory-mapped). In addition, the MMU provides access protection on a segment, block, or page basis. This chapter describes the specific hardware used to implement the MMU model of the OEA and the implementation-specific changes in the e600 core MMU model to support 36-bit physical addressing. Refer to Chapter 7, "Memory Management," in the *Programming Environments Manual* for a complete description of the conceptual model used for 32-bit physical addressing. Note that the e600 core does not implement the optional direct-store facility.

Two general types of memory accesses generated by processors that implement the PowerPC architecture require address translation—instruction accesses and data accesses generated by load and store instructions. In addition, the addresses specified by cache instructions and the optional external control instructions also require translation. Generally, the address translation mechanism is defined in terms of the segment descriptors and page tables that the processors use to locate the effective-to-physical address mapping for memory accesses. The segment information translates the effective address (EA) to an interim virtual address, and the page table information translates the virtual address (VA) to a physical address (PA).

The segment descriptors, used to generate the interim virtual addresses, are stored as segment registers on 32-bit implementations (such as the e600 core). In addition, two translation lookaside buffers (TLBs) are implemented to keep recently used page address translations within the core. Although the PowerPC OEA describes one MMU (conceptually), the e600 hardware maintains separate TLBs and table search resources for instruction and data accesses that can be performed independently (and simultaneously). Therefore, the e600 core is described as having two MMUs, one for instruction accesses (IMMU) and one for data accesses (DMMU).

The block address translation (BAT) mechanism is a software-controlled array that stores the available block address translations within the core. BAT array entries are implemented as pairs of BAT registers that are accessible as supervisor special-purpose registers (SPRs). There are separate instruction and data BAT mechanisms. In the e600 core, they reside in the instruction and data MMUs, respectively.

The MMUs, together with the interrupt processing mechanism, provide the necessary support for the operating system to implement a paged virtual memory environment and for enforcing protection of designated memory areas. Interrupt processing is described in [Chapter 4, "Interrupts."](#) [Section 4.3, "Interrupt Processing,"](#) describes the MSR that controls some of the critical functionality of the MMUs.

AltiVec Technology and the MMU Implementation

The AltiVec functionality in the e600 core affects the MMU model in the following ways:

- A data stream instruction (**dst[t]** or **dstst[t]**) can cause table search operations to occur after the instruction is retired.
- MMU interrupt conditions can cause a data stream operation to abort.
- Aborted VTQ-initiated table search operations can cause a line fetch skip.
- Execution of a **tlbsync** instruction can cancel an outstanding table search operation for a VTQ.

5.1 MMU Overview

The e600 core implements the memory management specification of the PowerPC OEA for 32-bit implementations but adds the capability to support 36-bit physical addressing. Thus it provides 4 Gbytes of effective address space accessible to supervisor and user programs, with a 4-Kbyte page size and 256-Mbyte segment size. In addition, the e600 MMUs use an interim virtual address (52 bits) and hashed page tables in the generation of 32-bit or 36-bit physical addresses (depending on the setting of `HID0[XAEN]`). Processors that implement the PowerPC architecture also have a BAT mechanism for mapping large blocks of memory. For the e600 core, block sizes range from 128 Kbytes to 4 Gbytes and are also software-programmable.

Basic features of the e600 core MMU implementation defined by the OEA are as follows:

- Support for real addressing mode—Effective-to-physical address translation can be disabled separately for data and instruction accesses.
- Block address translation—Each of the BAT array entries (four IBAT entries and four DBAT entries) provides a mechanism for translating blocks as large as 4 Gbytes from the 32-bit effective address space into the physical memory space. This can be used for translating large address ranges whose mappings do not change frequently. Four additional IBAT and DBAT entries are provided that can be enabled by setting `HID0[HIGH_BAT_EN]`, for a total of eight IBAT entries and eight DBAT entries.
- Segmented address translation—The 32-bit effective address is extended to a 52-bit virtual address by substituting 24 bits of upper address bits from the segment register. The 4 upper bits of the EA are used as an index into the segment register file. This 52-bit virtual address space is divided into 4-Kbyte pages, each of which can be mapped to a physical page.

The e600 core also provides the following features that are not required by the PowerPC architecture:

- Separate translation lookaside buffers (TLBs)—The 128-entry, two-way set-associative ITLBs and DTLBs keep recently used page address translations within the core.
- Table search operations performed in hardware—The 52-bit virtual address is formed and the MMU attempts to fetch the page table entry (PTE) that contains the physical address from the appropriate TLB within the core. If the translation is not found in either the BAT array or in a TLB (that is, a TLB miss occurs), the hardware performs a table search operation (using a hashing function) to search for the PTE. Hardware table searching is the default mode for the e600 core; however, if `HID0[STEN] = 1`, software table searching is performed.
- Table search operations performed in software—The e600 core also supports software table searching (when `HID0[STEN] = 1`) for TLB misses. In this case, the `TLBMISS` register saves the

effective address of the access that requires a software table search. The PTEHI and PTELO registers, as well as the **tlbli** and **tlbld** instructions are resources used in reloading the TLBs during a software table search operation. Also three interrupts are used to support software table searching when $HID0[STEN] = 1$ and a TLB miss occurs. They are as follows:

- For an instruction fetch, an ITLB miss interrupt,
- For a data load, a DTLB miss-on-load interrupt,
- For a data store, a DTLB miss-on-store interrupt
- TLB invalidation—The e600 core implements the optional TLB invalidate entry (**tlbie**) and TLB synchronize (**tlbsync**) instructions that can be used to invalidate TLB entries. For more information on the **tlbie** and **tlbsync** instructions, see [Section 5.4.4.2, “TLB Invalidation.”](#)
- Extended 36-bit physical addresses provide for 64 Gbytes of physical memory when $HID0[XAEN]$ is set.

[Table 5-1](#) summarizes the e600 core MMU features, including those defined by the PowerPC architecture (OEA) for 32-bit processors and those specific to the e600 core.

Table 5-1. MMU Features Summary

Feature Category	Architecturally Defined/ e600-Specific	Feature
Address ranges	Architecturally defined	2^{32} bytes of effective address
		2^{52} bytes of virtual address
		2^{32} bytes of physical address
	e600-specific	Optional 2^{36} bytes of physical address
Page size	Architecturally defined	4 Kbytes
Segment size	Architecturally defined	256 Mbytes
Block address translation	e600-specific	Range of 128 Kbyte–4 Gbyte block sizes
		Eight IBAT and eight DBAT entries in BAT array
Memory protection	Architecturally defined	Segments selectable as no-execute
		Pages selectable as user/supervisor and read-only or guarded
		Blocks selectable as user/supervisor and read-only or guarded
Page history	Architecturally defined	Referenced and changed bits defined and maintained
Page address translation	Architecturally defined	Translations stored as PTEs in hashed page tables in memory
		Page table size determined by mask in SDR1 register
TLBs	Architecturally defined	Instructions for maintaining TLBs (tlbie and tlbsync instructions in the e600 core)
	e600-specific	128-entry, two-way set associative ITLB 128-entry, two-way set associative DTLB LRU replacement algorithm
Segment descriptors	Architecturally defined	Stored as segment registers within the core (two identical copies maintained)

Table 5-1. MMU Features Summary (continued)

Feature Category	Architecturally Defined/ e600-Specific	Feature
Page table search support—Hardware	e600-specific	The e600 core can perform the table search operation in hardware (or software, as listed below).
Page table search support—Software	e600-specific	TLBMISS register (missed effective address) PTEHI and PTELO registers (contents of corresponding PTE)
		Three MMU interrupts, defined: ITLB miss interrupt, DTLB miss on load interrupt, and DTLB miss on store (or store and C = 0) interrupt; MMU-related bits are set in SRR1 for these interrupts
		tibli rB instruction for loading ITLB entries tlbid rB instruction for loading DTLB entries

5.1.1 Memory Addressing

A program references memory using the effective (logical) address computed by the processor when it executes a load, store, branch, or cache instruction, and when it fetches the next instruction. The effective address is translated to a physical address according to the procedures described in Chapter 7, “Memory Management,” in *The Programming Environments Manual*, augmented with information in this chapter. The core memory subsystem uses the physical address for the access.

For a complete discussion on effective address calculation, see [Section 2.3.2.3, “Effective Address Calculation.”](#)

5.1.2 MMU Organization

[Figure 5-1](#) shows the conceptual organization of a PowerPC MMU in a 32-bit implementation that generates 32-bit physical addresses. Note that it does not describe the specific hardware used to implement the memory management function for a particular processor. Processors may optionally implement TLBs within the core, hardware support for the automatic search of the page tables for PTEs, and other hardware features (invisible to the system software) not shown. Also, the e600 core generates a 36-bit physical address, which is not represented by the 32-bit physical address in [Figure 5-1](#).

The instruction addresses are generated by the processor for sequential instruction fetches and addresses that correspond to a change of program flow. Data addresses are generated by load, store, and cache instructions.

As shown in [Figure 5-1](#), when the default 32-bit physical addresses are generated, the high-order bits of the effective address, EA[0–19] (or a smaller set of address bits, EA[0–*n*], in the case of blocks), are translated into physical address bits PA[0–19]. The low-order address bits, EA[20–31], are untranslated and are therefore identical for both effective and physical addresses. After translating the address, the MMU passes the resulting 32-bit physical address to the core memory subsystem.

The MMUs record whether the translation is for an instruction or data access, whether the processor is in user or supervisor mode and, for data accesses, whether the access is a load or a store operation. The MMUs use this information to appropriately direct the address translation and to enforce the protection

hierarchy programmed by the operating system. Section 4.3, “Interrupt Processing,” describes the MSR that controls some of the critical functionality of the MMUs.

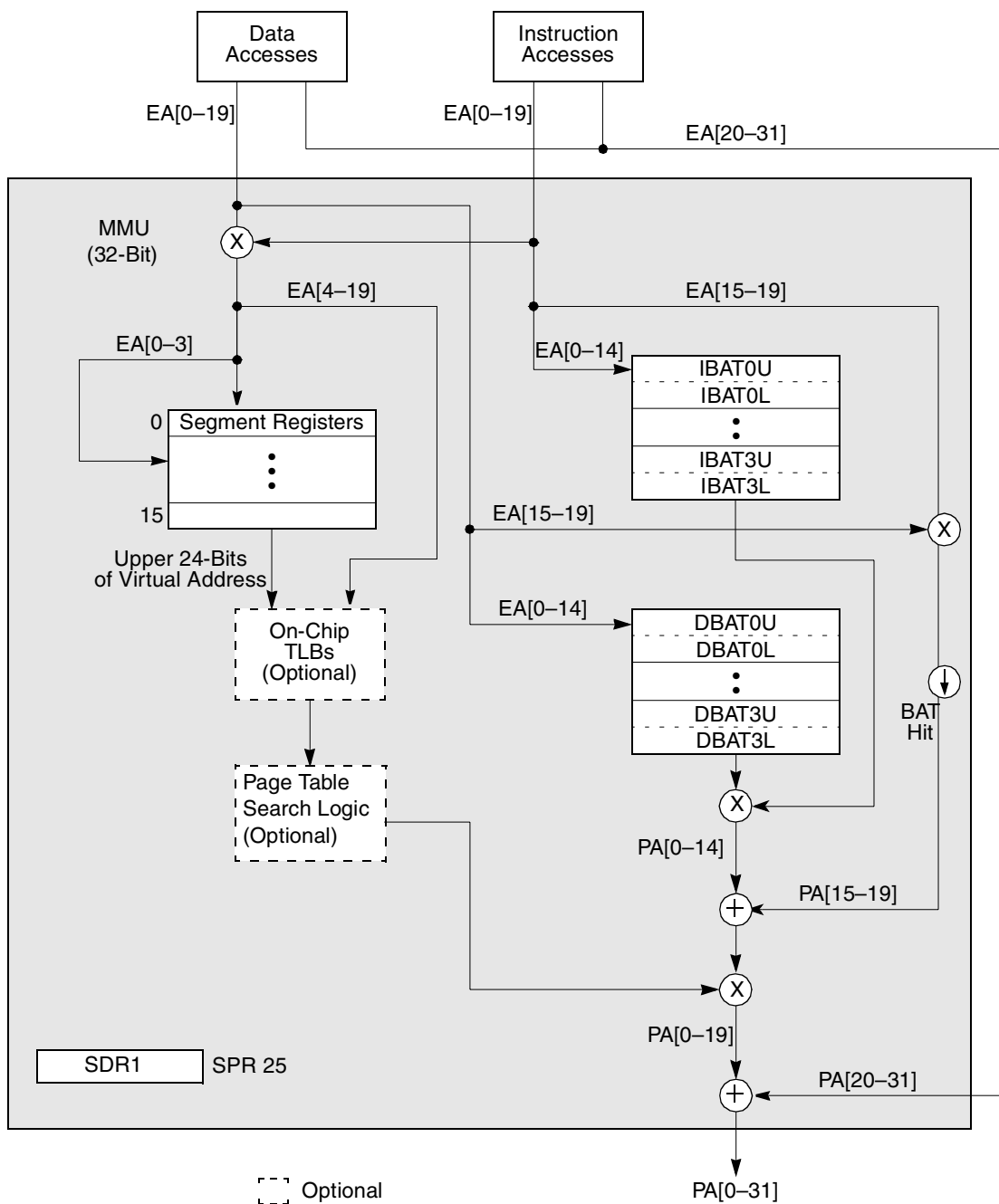


Figure 5-1. MMU Conceptual Block Diagram for a 32-Bit Physical Address (Not the e600 Core)

Figure 5-2 and Figure 5-3 contain the block diagrams of the IMMU and DMMU of the e600 core and show how a 36-bit physical address is generated. Address bits EA[20-26] index into the instruction and data caches on the e600 core to select a cache set. The remaining physical address bits are then compared with the tag fields (comprised of bits PA[0-23]) of the two selected cache blocks to determine if a cache hit has

occurred. In the case of a cache miss, the instruction or data access is forwarded to the L2 cache tags to check for an L2 cache hit. In case of a miss, the access is forwarded to the bus interface unit (BIU).

Figure 5-2 and Figure 5-3 also show the two TLBs maintained by the e600 core, which have the following characteristics:

- 128 entries, two-way set associative (64 x 2), LRU replacement
- Hardware or software table search operations and TLB reloads
- Hardware or software update of referenced (R) and changed (C) bits in the translation table
- 36-bit physical addresses

In the event of a TLB miss, the TLB entry must be loaded. The TLB is loaded automatically by the hardware or by the software table search algorithm, depending on the HID0[STEN] setting.

Figure 5-2 and Figure 5-3 show the detailed routing of addresses that are generated by the IMMU and DMMU respectively when 36-bit addressing (extended addressing) is used. In this case, EA[0–19] (or a smaller subset EA[0–n], in the case of blocks) are translated into physical address bits PA[0–23] and the low-order address bits, EA[20–31] are untranslated, but shifted down to comprise PA[24–35]. Also, in this case, EA[20–26] index into the caches within the core so that PA[0–23] from the MMU can be compared with the tag fields (comprised of PA[0–23]) to determine if a cache hit has occurred.

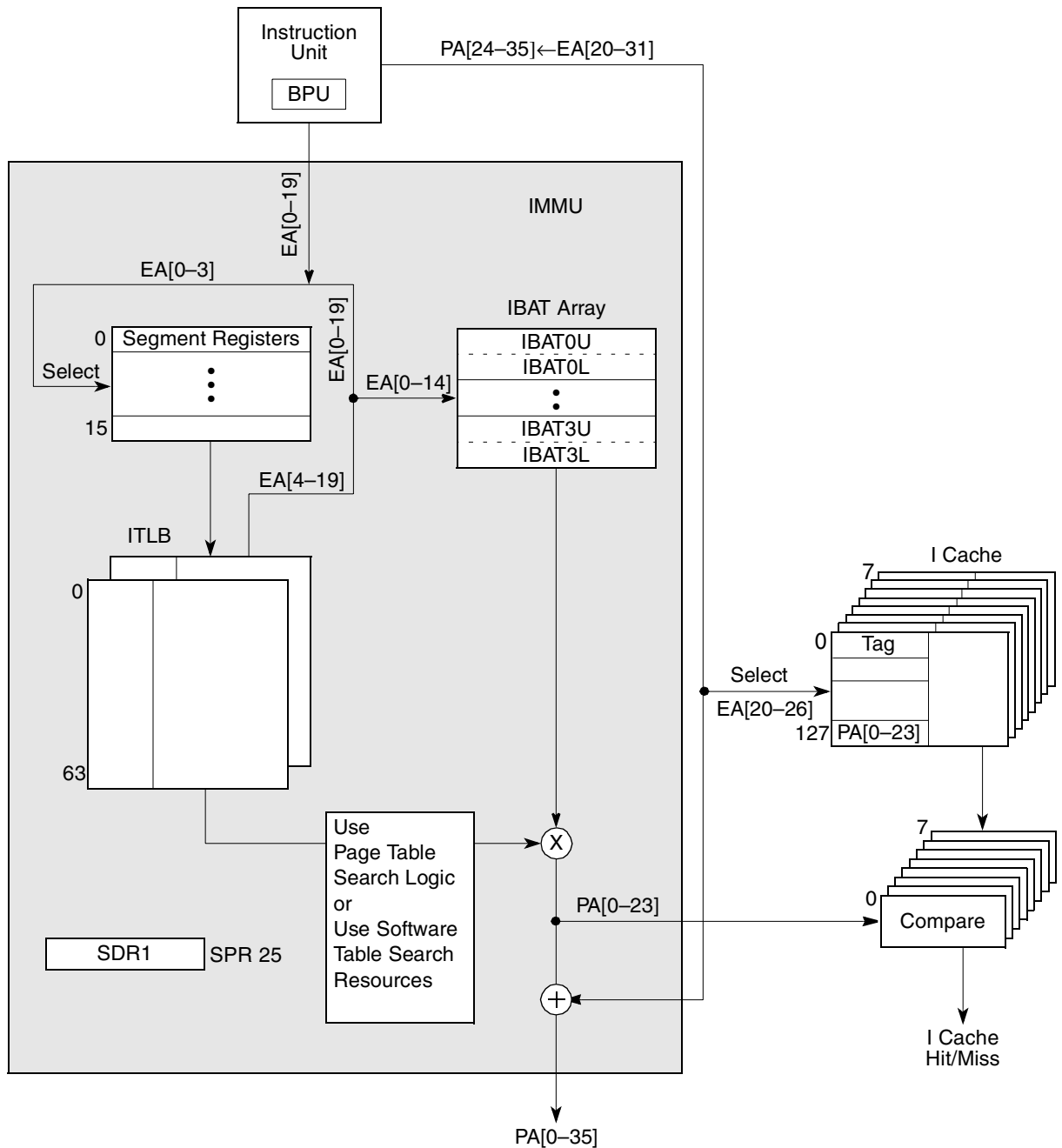


Figure 5-2. e600 Core IMMU Block Diagram, 36-Bit Physical Addressing

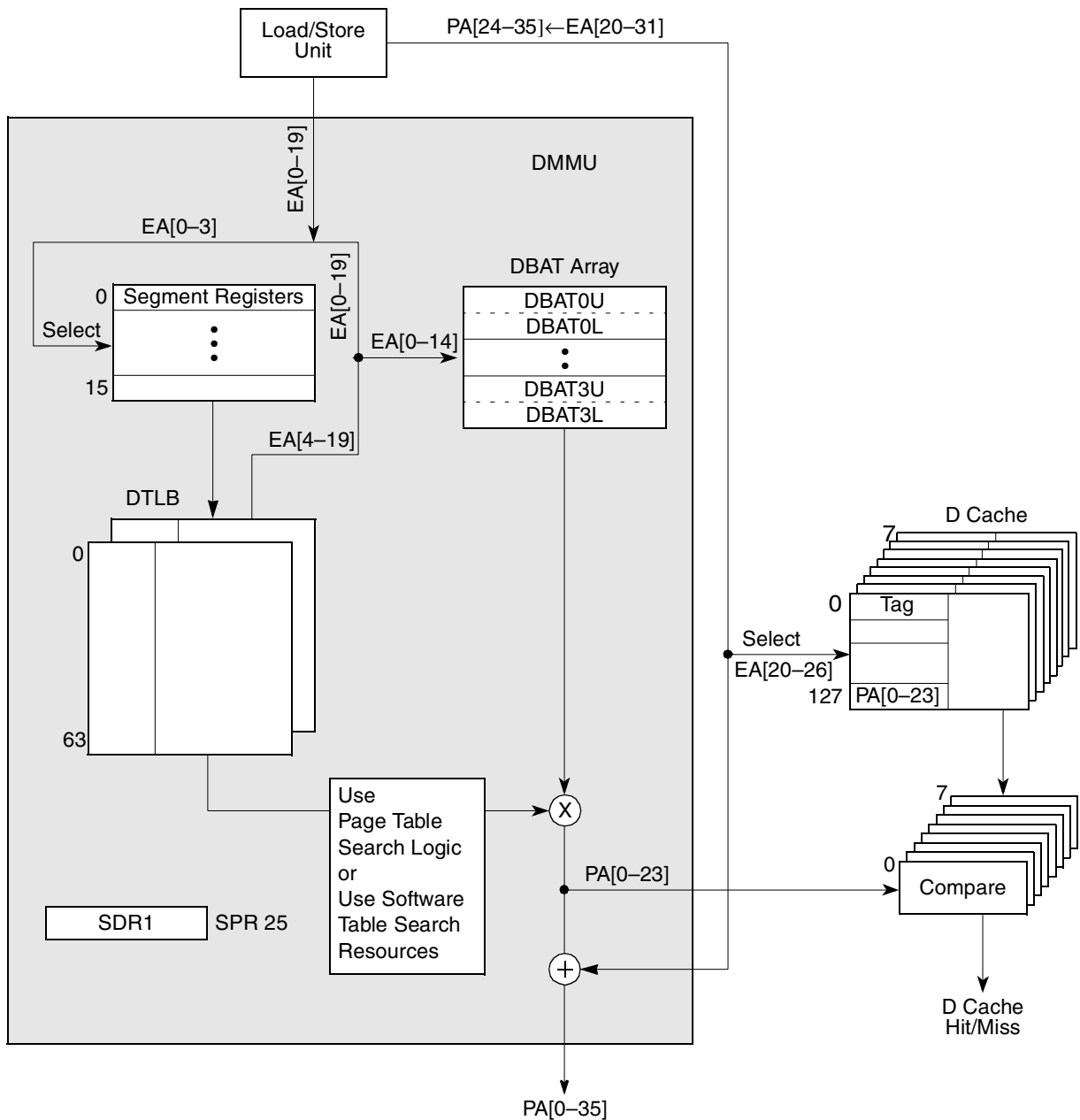


Figure 5-3. e600 Core DMMU Block Diagram, 36-Bit Physical Addressing

Figure 5-4 shows the detailed routing of addresses for the e600 core that are generated by the DMMU when 36-bit addressing (extended addressing) is used. Also the extended block size is enabled so that the EA[0-19] is translated into physical address bits PA[0-23] and the low-order address bits, EA[20-31], are untranslated but shifted down to comprise PA[24-35]. Also, in this case, additional BATs are available (DBAT4U to DBAT7L) for use. The same features, extended block size and additional BATs, would be generated by the IMMU as well.

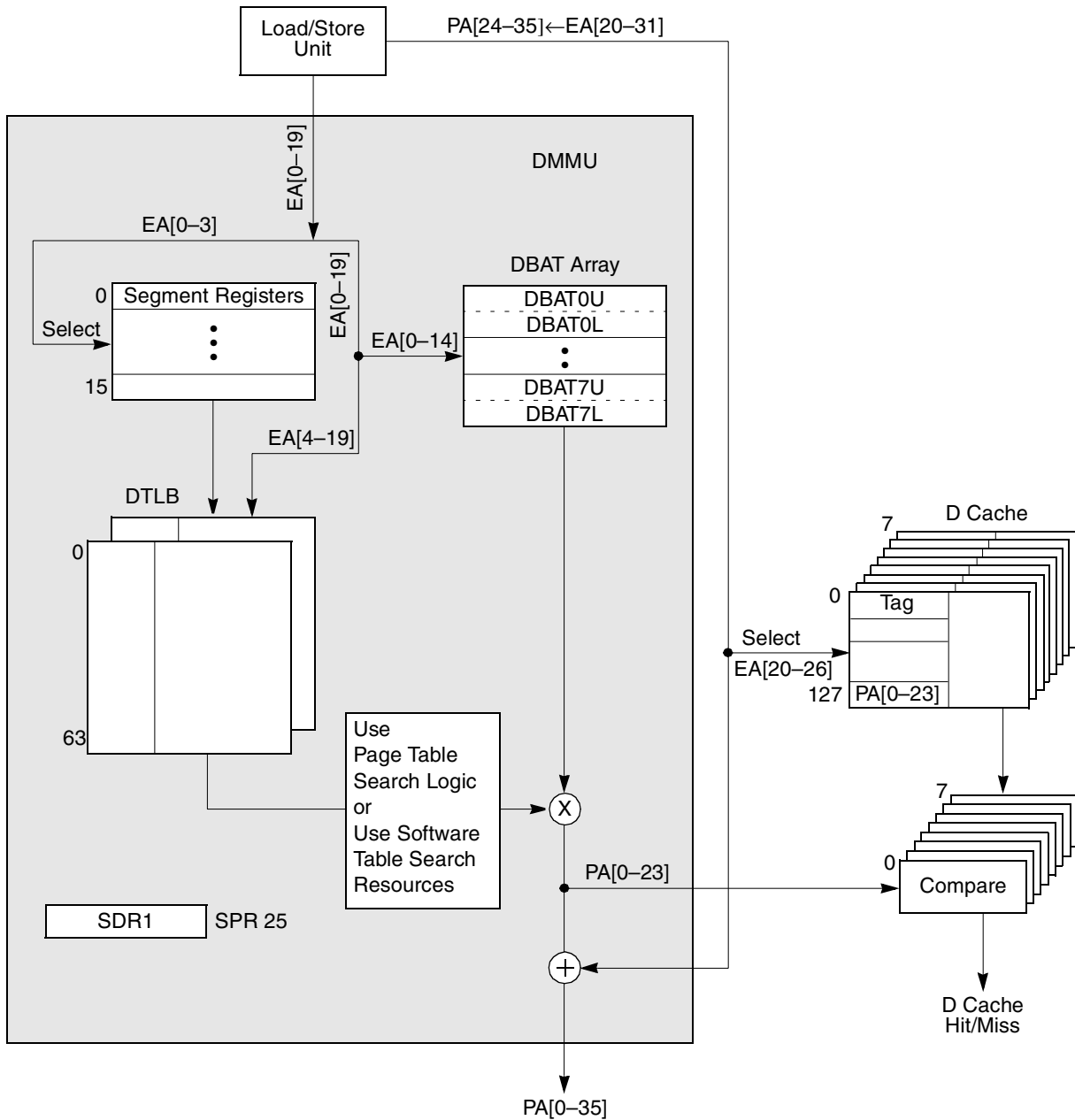


Figure 5-4. e600 Core DMMU Block Diagram with Extended Block Size and Additional BATs

5.1.3 Address Translation Mechanisms

Processors that implement the PowerPC architecture support the following types of address translation:

- Page address translation—Translates the page frame address for a 4-Kbyte page
- Block address translation—Translates the block number for blocks that range in size from 128 Kbytes to 4 Gbytes
- Real addressing mode—Address translation is disabled; therefore, no translation is done and the physical address is identical to the effective address.

Figure 5-5 shows the three address translation mechanisms provided by the MMUs for 32-bit physical addressing and Figure 5-6 shows the same mechanism for 36-bit physical addressing. The segment descriptors shown in the figures control the page address translation mechanism. When an access uses page address translation, the appropriate segment descriptor is required. The appropriate segment descriptor is selected from the 16 segment registers on the core by the four highest-order effective address bits.

A control bit in the corresponding segment descriptor then determines if the access is to memory (memory-mapped) or to the direct-store interface space ($SR_n[T]$). Note that the direct-store interface was present in the architecture only for compatibility with existing I/O devices that used this interface. The e600 core does not support the direct-store interface ($SR_n[T] = 1$). When an access is determined to be to the direct-store interface space, the e600 core takes a DSI interrupt if it is a data access (see Section 4.6.3, “DSI Interrupt (0x00300)”), and takes an ISI interrupt if it is an instruction access (see Section 4.6.4, “ISI Interrupt (0x00400)”).

For memory accesses translated by a segment descriptor, the interim virtual address is generated using the information in the segment descriptor. Page address translation corresponds to the conversion of this virtual address into the 32-bit or 36-bit physical address used by the core memory subsystem. In most cases, the physical address for the page resides in a TLB on the core and is available for quick access. However, if the page address translation misses in the TLB in the core, the MMU causes a search of the page tables in memory. Page tables can be searched by hardware using the virtual address information and a hashing function to locate the required physical address or the e600 core vectors to interrupt handlers that use software to search the page tables (if $HID0[STEN] = 1$).

Because blocks are larger than pages, there are fewer higher-order effective address bits to be translated into physical address bits (more low-order address bits (at least 17) are untranslated to form the offset into a block) for block address translation. Also, instead of segment descriptors and a TLB, block address translations use the BAT registers in the e600 core as a BAT array. If an effective address matches the corresponding field of a BAT register, the information in the BAT register is used to generate the physical address; in this case, the results of the page translation (occurring in parallel) are ignored.

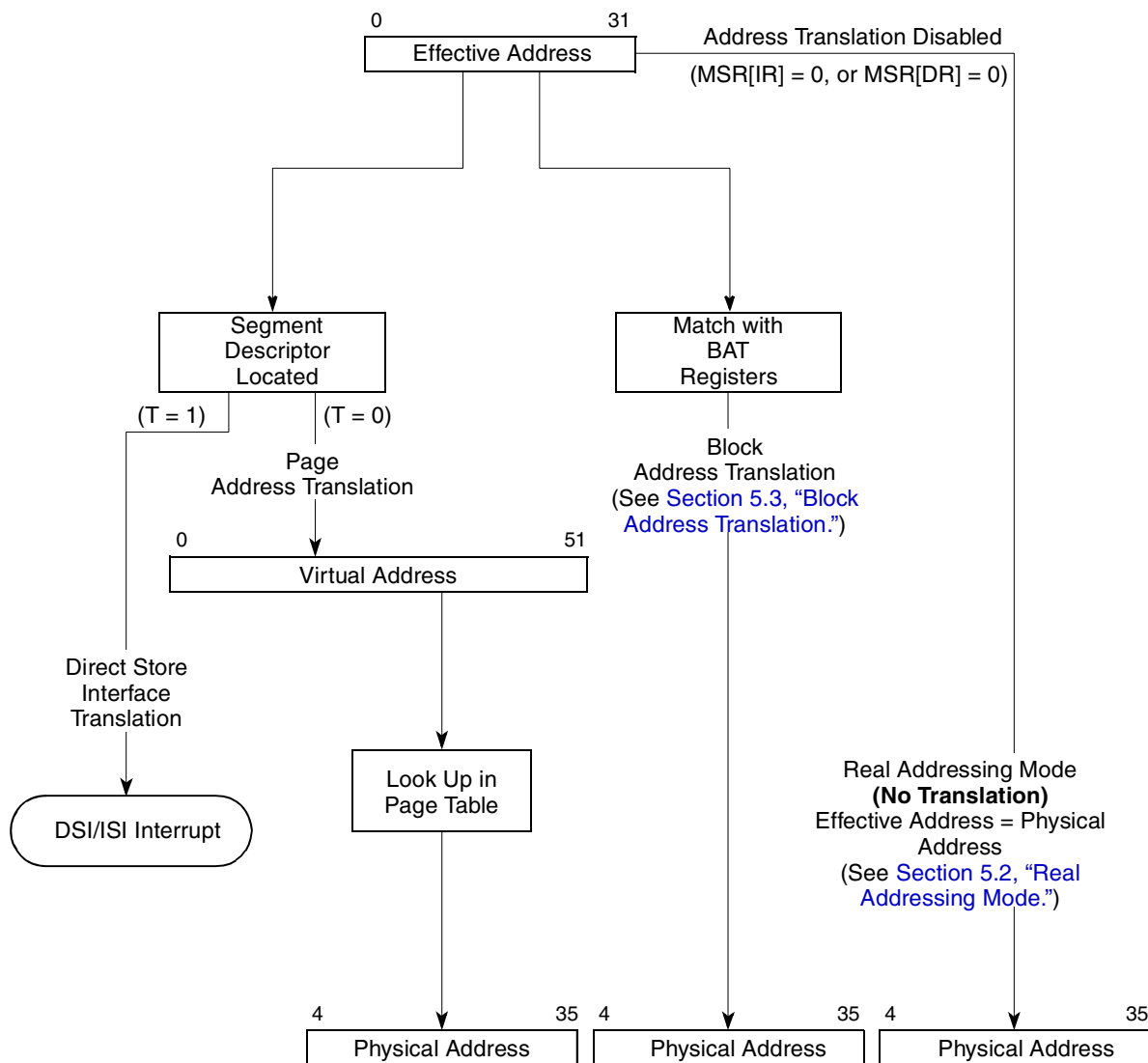


Figure 5-5. Address Translation Types for 32-Bit Physical Addressing

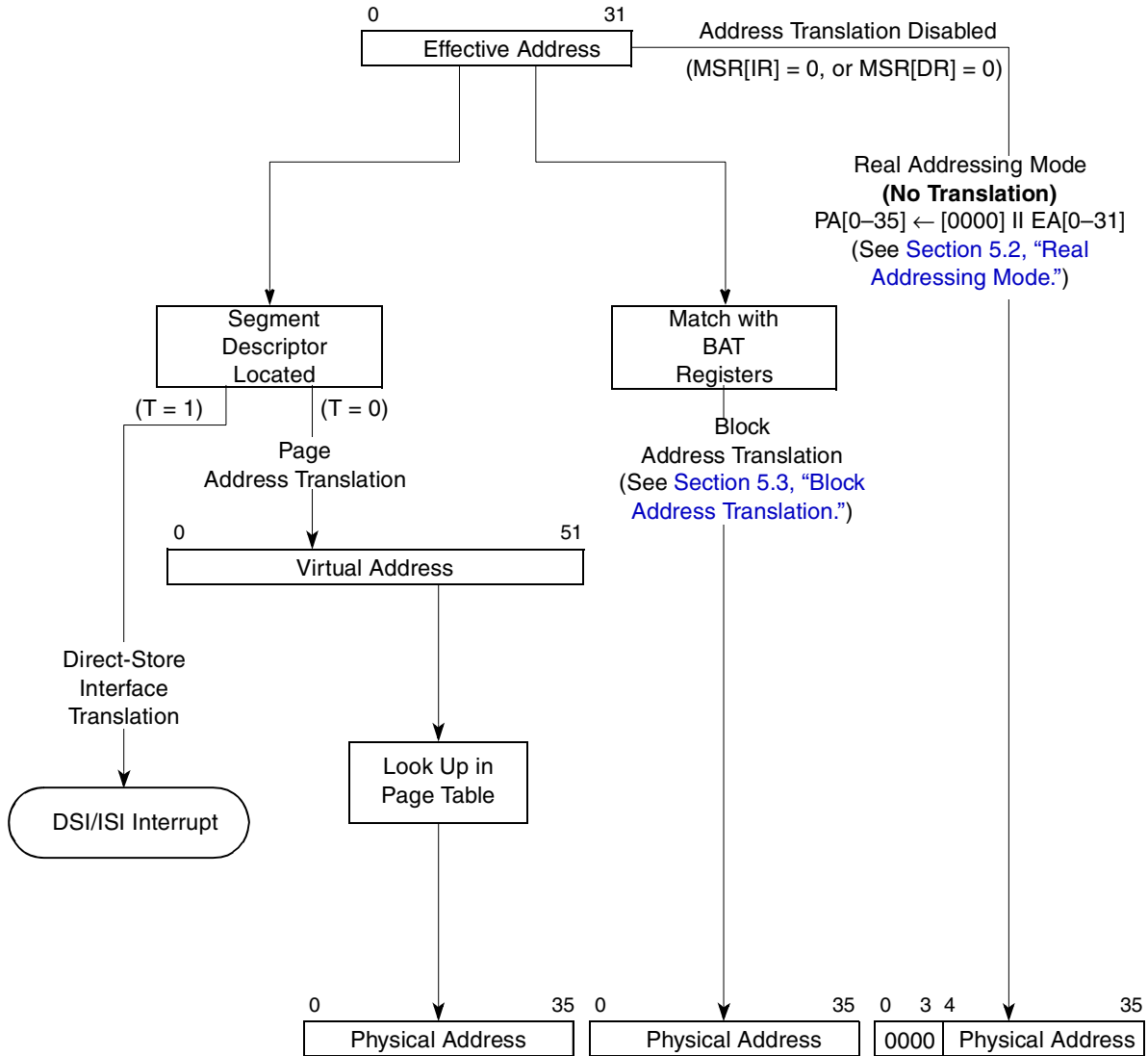


Figure 5-6. Address Translation Types for 36-Bit Physical Addressing

When the processor generates an access, and the corresponding address translation enable bit in MSR is cleared ($MSR[IR] = 0$ or $MSR[DR] = 0$), the resulting physical address is identical to the effective address and all other translation mechanisms are ignored. Instruction address translation and data address translation are enabled by setting $MSR[IR]$ and $MSR[DR]$, respectively.

When extended addressing is enabled, $HID0[XAEN] = 1$, and the corresponding address translation bit in MSR is cleared ($MSR[IR] = 0$ or $MSR[DR] = 0$), the 36-bit physical address is formed by concatenating four leading zeros to the 32-bit effective address.

5.1.4 Memory Protection Facilities

In addition to the translation of effective addresses to physical addresses, the MMUs provide access protection of supervisor areas from user access and can designate areas of memory as read-only as well as no-execute or guarded. Table 5-2 shows the protection options supported by the MMUs for pages.

Table 5-2. Access Protection Options for Pages

Option	User Read		User Write	Supervisor Read		Supervisor Write
	I-Fetch	Data		I-Fetch	Data	
Supervisor-only	—	—	—	√	√	√
Supervisor-only-no-execute	—	—	—	—	√	√
Supervisor-write-only	√	√	—	√	√	√
Supervisor-write-only-no-execute	—	√	—	—	√	√
Both (user/supervisor)	√	√	√	√	√	√
Both (user-/supervisor) no-execute	—	√	√	—	√	√
Both (user-/supervisor) read-only	√	√	—	√	√	—
Both (user/supervisor) read-only-no-execute	—	√	—	—	√	—

√ Access permitted
 — Protection violation

The no-execute option provided in the segment register lets the operating system program determine whether instructions can be fetched from an area of memory. The remaining options are enforced based on a combination of information in the segment descriptor and the page table entry. Thus the supervisor-only option allows only read and write operations generated while the processor is operating in supervisor mode ($MSR[PR] = 0$) to access the page. User accesses that map into a supervisor-only page cause an interrupt.

Finally, a facility in the VEA and OEA allows pages or blocks to be designated as guarded, preventing out-of-order accesses that may cause undesired side effects. For example, areas of the memory map used to control I/O devices can be marked as guarded so accesses do not occur unless they are explicitly required by the program.

For more information on memory protection, see the section, “Memory Protection Facilities,” in Chapter 7, “Memory Management,” in the *The Programming Environments Manual*.

5.1.5 Page History Information

The MMUs of processors that support the PowerPC architecture also define referenced (R) and changed (C) bits in the page address translation mechanism that can be used as history information relevant to the page. The operating system can use these bits to determine which areas of memory to write back to disk when new pages must be allocated in main memory. While these bits are initially programmed by the operating system into the page table, the architecture specifies that they can be maintained either by the processor hardware (automatically) or by some software-assist mechanism.

When loading the TLBs in hardware, the e600 core checks the state of the changed and referenced bits for the matched PTE. If the referenced bit is not set and the table search operation is initially caused by a load operation or by an instruction fetch, the core automatically sets the referenced bit in the translation table. Similarly, if the table search operation is caused by a store operation and either the referenced bit or the changed bit is not set, the hardware automatically sets both bits in the translation table. In addition, when the address translation of a store operation hits in the DTLB, the core checks the state of the changed bit. If the bit is not already set, the hardware automatically updates the DTLB and the translation table in memory to set the changed bit. For more information, see [Section 5.4.2, “Page History Recording.”](#)

When software table searching is enabled ($HID0[STEN] = 1$), the software table search routines used by the e600 core can set the R bit when a PTE is accessed. Also, the core causes an interrupt (to vector to the software table search routines) when the C bit in the TLB is cleared but a store occurs, allowing the corresponding PTE to be updated by software.

5.1.6 General Flow of MMU Address Translation

The following sections describe the general flow used by processors that implement the PowerPC architecture, to translate effective addresses to physical addresses. There are three types of addressing translations used by the PowerPC architecture, page address, block address, and real addressing mode. Two sizes of physical addresses, 32-bit or 36-bit, can be generated depending on whether extended addressing is enabled ($HID0[XAEN] = 1$). Details for how an effective address is translated to a 32-bit physical address are described in Chapter 7, “Memory Management,” in the *The Programming Environments Manual*. The following sections describe the differences in address translation for an extended physical address (36-bits).

5.1.6.1 Real Addressing Mode and Block Address Translation Selection

When an instruction or data access is generated and the corresponding instruction or data translation is disabled ($MSR[IR] = 0$ or $MSR[DR] = 0$), real addressing mode is used (physical address equals effective address) and the access continues to the core memory subsystem as described in [Section 5.2, “Real Addressing Mode.”](#)

Figure 5-7 shows the flow the MMUs use in determining which translation to select: real addressing mode, block address, or page address.

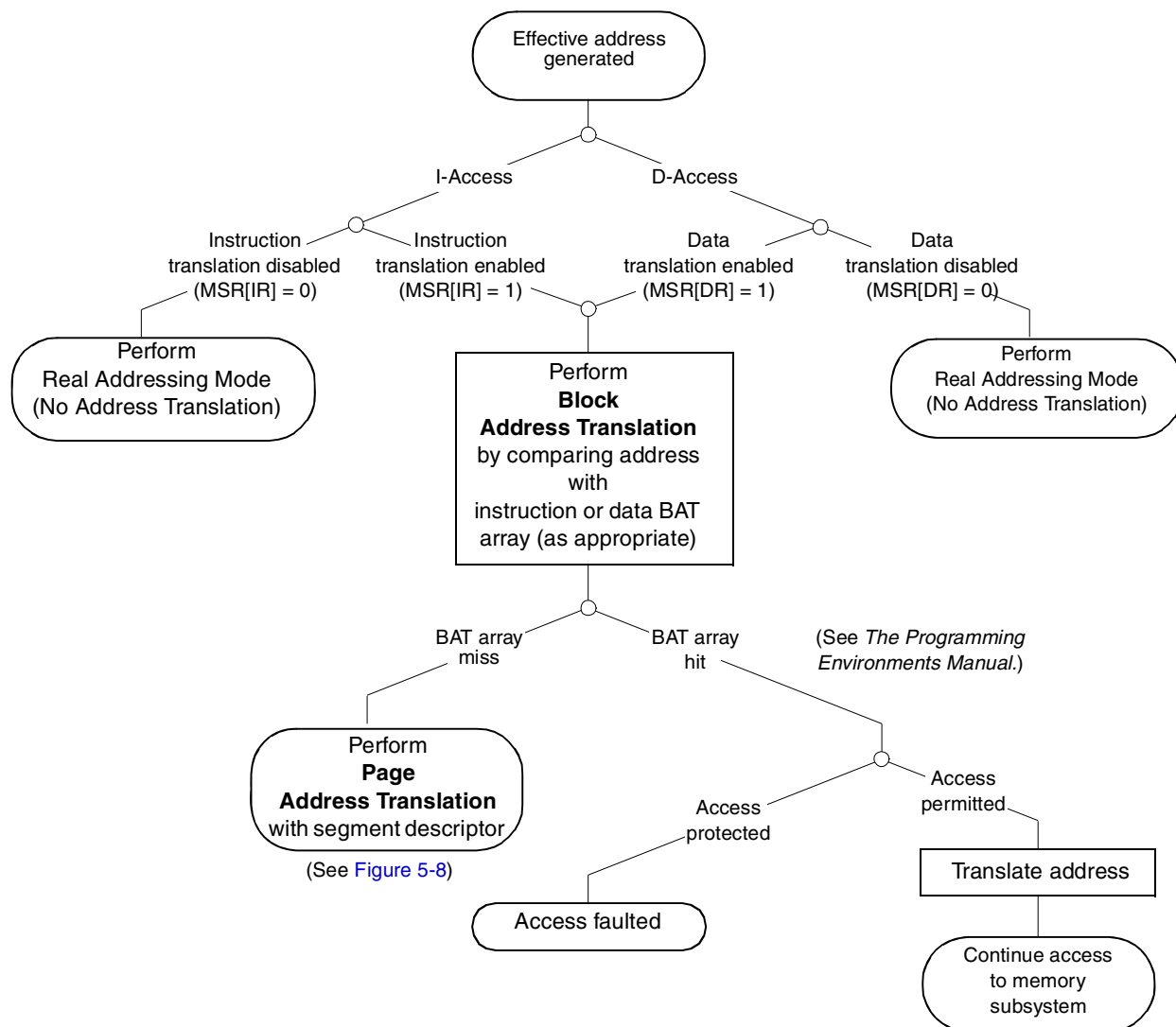


Figure 5-7. General Flow in Selection of Which Address Translation to Use

Note that if the BAT array search results in a hit, the access is qualified with the appropriate protection bits. If the access violates the protection mechanism, an interrupt (ISI or DSI) is generated.

5.1.6.2 Page Address Translation Selection

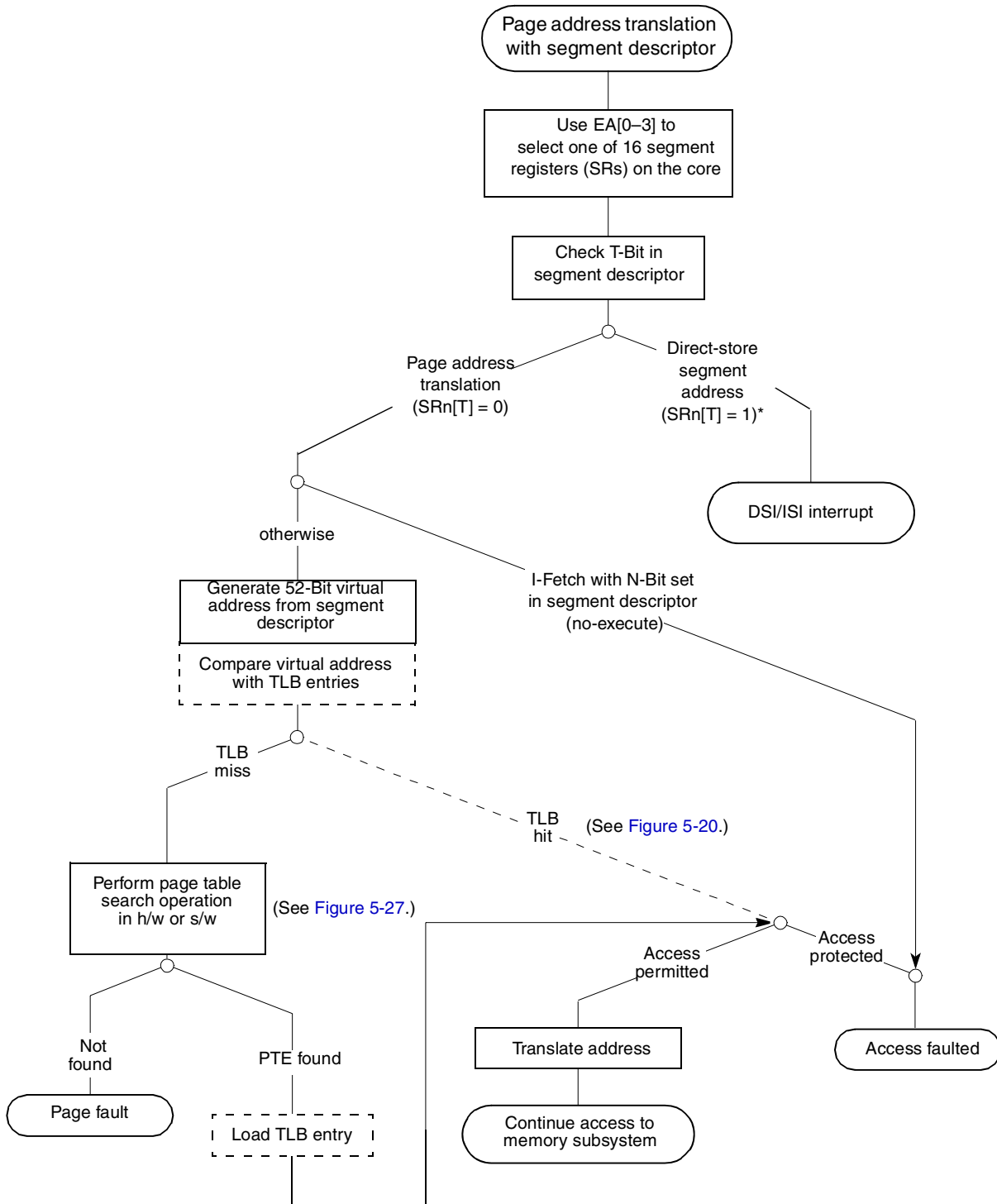
If address translation is enabled and the effective address information does not match a BAT array entry, the segment descriptor must be located. When the segment descriptor is located, the T bit in the segment descriptor selects whether the translation is to a page or to a direct-store segment as shown in Figure 5-8. The segment descriptor for an access is contained in one of 16 segment registers in the core; effective address bits EA[0–3] select one of the 16 segment registers.

Note that the e600 core does not implement the direct-store interface, and accesses to these segments cause a DSI or ISI interrupt. In addition, [Figure 5-8](#) also shows the way in which the no-execute protection is enforced; if the N bit in the segment descriptor is set and the access is an instruction fetch, the access is faulted as described in [Section 5.4.3, “Page Memory Protection.”](#) Note that the figure shows the flow for these cases as described by the PowerPC OEA, and so the TLB references are shown as optional. Because the e600 core implements TLBs, these branches are valid and are described in more detail throughout this chapter.

If $SR[T] = 0$, page address translation is selected. The information in the segment descriptor is then used to generate the 52-bit virtual address. The virtual address is then used to identify the page address translation information (stored as page table entries (PTEs) in a page table in memory). For increased performance, the e600 core has two TLBs to cache recently used translations within the core.

If an access hits in the appropriate TLB, page translation succeeds and the physical address bits are forwarded to the core memory subsystem. If the required translation is not resident, the MMU performs a search of the page table. In this case, the e600 core either initiates a search of the page table in hardware or traps to one of three interrupt handlers for the system software to perform the page table search (if $HIDO[STEN] = 1$). If the required PTE is found, a TLB entry is allocated and the page translation is attempted again. This time, the TLB is guaranteed to hit. When the translation is located, the access is qualified with the appropriate protection bits. If the access causes a protection violation, an ISI or DSI interrupt is generated.

If the PTE is not found by the table search operation, a page fault condition exists and an ISI or DSI interrupt occurs so software can handle the page fault.



- - - - Optional to the PowerPC architecture. Implemented in the e600 core.

* In the case of instruction accesses, causes ISI interrupt.

Figure 5-8. General Flow of Page Translation

5.1.7 MMU Interrupts Summary

To complete any memory access, the effective address must be translated to a physical address. As specified by the architecture, an MMU interrupt condition occurs if this translation fails for one of the following reasons:

- Page fault—There is no valid entry in the page table for the page specified by the effective address (and segment descriptor) and there is no valid BAT translation.
- An address translation is found but the access is not allowed by the memory protection mechanism.

Additionally, because the e600 core can use software to perform table search operations, the processor also takes an interrupt when $HID0[STEN] = 1$ and:

- There is a miss in the corresponding (instruction or data) TLB, or
- The page table requires an update to the changed (C) bit.

The state saved by the processor for each of these interrupts contains information that identifies the address of the failing instruction. Refer to [Chapter 4, “Interrupts,”](#) for a more detailed description of interrupt processing.

When software table searching is selected, a page fault condition (PTE not found in the page tables in memory) is detected by the software that performs the table search operation (and not the e600 core hardware). Therefore, it does not cause an e600 core interrupt in the strictest sense, in that interrupt processing as described in [Chapter 4, “Interrupts,”](#) does not occur. However, in order to maintain architectural compatibility with software written for other devices that implement the PowerPC architecture, the software that detects this condition should synthesize an interrupt by setting the appropriate bits in the DSISR or SRR1 and branching to the ISI or DSI interrupt handler. Refer to [Section 5.5.5, “Implementation-Specific Software Table Search Operation,”](#) for more information and examples of this interrupt software. The remainder of this chapter assumes that the table search software emulates this interrupt and refers to this condition as an interrupt.

The translation interrupt conditions defined by the OEA for 32-bit implementations cause either the ISI or the DSI interrupt to be taken as shown in [Table 5-3](#).

Table 5-3. Translation Interrupt Conditions

Condition	Description	Interrupt
Page fault (no PTE found)	No matching PTE found in page tables (and no matching BAT array entry)	I access: ISI ¹ SRR1[1] = 1
		D access: DSI ¹ DSISR[1] = 1
Block protection violation	Conditions described for block in “Block Memory Protection” in Chapter 7, “Memory Management,” in <i>The Programming Environments Manual</i> .”	I access: ISI SRR1[4] = 1
		D access: DSI DSISR[4] = 1
Page protection violation	Conditions described for page in “Page Memory Protection” in Chapter 7, “Memory Management,” in <i>The Programming Environments Manual</i> .	I access: ISI ² SRR1[4] = 1
		D access: DSI ² DSISR[4] = 1

Table 5-3. Translation Interrupt Conditions (continued)

Condition	Description	Interrupt
No-execute protection violation	Attempt to fetch instruction when SR[N] = 1	ISI SRR1[3] = 1
Instruction fetch from direct-store segment	Attempt to fetch instruction when SR[T] = 1	ISI SRR1[3] = 1
Data access to direct-store segment (including floating-point accesses)	Attempt to perform load or store (including FP load or store) when SR[T] = 1	DSI DSISR[5] = 1
Instruction fetch from guarded memory	Attempted instruction fetch, MSR[IR] = 1, and either matching xBAT[G] = 1, or no matching BAT entry and PTE[G] = 1	ISI SRR1[3] = 1

¹ The e600 core hardware vectors to these interrupts automatically when HID0[STEN] = 0. When HID0[STEN] = 1, it is assumed that the software that performs the table search operations vectors to these interrupts and sets the appropriate bits when a page fault condition occurs.

² The table search software can also vector to these interrupt conditions.

In addition to the translation exceptions, there are other MMU-related conditions (some defined as implementation-specific) that can cause an interrupt to the e600 core. These exceptions map to processor interrupts as shown in [Table 5-4](#). For example, the core also defines three exception conditions to support software table searching. The only exception conditions that occur when MSR[DR] = 0 are those that cause an alignment interrupt for data accesses.

For more detailed information about the conditions that cause an alignment interrupt (in particular for string/multiple instructions), see [Section 4.6.6, “Alignment Interrupt \(0x00600\).”](#)

Note that some interrupt conditions depend upon whether the memory area is set up as write-through ($W = 1$) or cache-inhibited ($I = 1$). These bits are described fully in the section titled “Memory/Cache Access Attributes,” in Chapter 5, “Cache Model and Memory Coherency,” of *The Programming Environments Manual*. Refer to [Chapter 4, “Interrupts,”](#) in this book and to Chapter 6, “Interrupts,” in *The Programming Environments Manual* for a complete description of the SRR1 and DSISR bit settings for these interrupts.

Even though, for data accesses, the e600 core LSU initiates out-of-order accesses, the MMU prevents the changed bit in the PTE from being updated erroneously in these cases; but the LRU algorithm is updated. The MMU does not initiate interrupt processing for any exceptions until the instruction that caused the interrupt is the next to be retired. Also, the e600 core MMU does not initiate a search operation due to a TLB miss (including misses for **dcbt**, **dst**, and **dstst**) until the request is required by the program flow.

Table 5-4. Other MMU Exception Conditions

Exception Condition	Description	Interrupt
TLB miss for an instruction fetch (HID0[STEN] = 1)	No matching entry found in IBAT or ITLB	ITLB miss interrupt. For details on bits set for this interrupt, see Section 4.6.15, “TLB Miss Interrupts.”
TLB miss for a data load access (HID0[STEN] = 1)	No matching entry found in DBAT or DTLB for data load access	DTLB miss on load interrupt For details on bits set for this interrupt, see Section 4.6.15, “TLB Miss Interrupts.”
TLB miss for a data store access, or data store access and C = 0 (HID0[STEN] = 1)	No matching entry found in DBAT or DTLB for data store access, or matching DLTB entry has C = 0 and the PTE’s C bit must be set due to a data store operation	DTLB miss on store interrupt SRR1[11] = 0 For details on bits set for this interrupt, see Section 4.6.15, “TLB Miss Interrupts.”
		DTLB hit on store interrupt with data store access and C = 0 SRR1[11] = 1 For details on bits sets during the interrupt, see Section 4.6.15, “TLB Miss Interrupts.”
dcbz with W = 1 or I = 1	dcbz instruction to write-through or cache-inhibited segment or block	Alignment interrupt (not required by architecture for this condition)
lwarx or stwcx. to direct-store segment	Reservation instruction or external control instruction when SR[T] = 1	DSI interrupt DSISR[5] = 1
Floating-point load or store to direct-store segment	FP memory access when SR[T] = 1	See data access to direct-store segment in Table 5-3.
Load or store that results in a direct-store error	Does not occur in the e600 core	Does not apply
eciwx or ecowx attempted when external control facility disabled	eciwx or ecowx attempted with EAR[E] = 0	DSI interrupt DSISR[11] = 1
lmw , stmw , lswi , lswx , stswi , or stswx instruction attempted in little-endian mode	lmw , stmw , lswi , lswx , stswi , or stswx instruction attempted while MSR[LE] = 1	Alignment interrupt
Operand misalignment	Translation enabled and a stmw , stwcx. , lmw , lwarx or floating-point load/store operand is not word-aligned	Alignment interrupt (some of these cases are implementation-specific). See Section 2.2.3, “Alignment and Misaligned Accesses.”

5.1.8 MMU Instructions and Register Summary

The MMU instructions and registers allow the operating system to set up the block address translation areas and the page tables in memory.

Note that because the implementation of TLBs is optional, the instructions that refer to these structures are also optional in the architecture. However, as these structures serve as caches of the page table, the architecture specifies a software protocol for maintaining coherency between these caches and the tables in memory whenever the tables in memory are modified. When the tables in memory are changed, the operating system purges these caches of the corresponding entries, allowing the translation caching mechanism to refetch from the tables when the corresponding entries are required.

Note that the e600 core implements all TLB-related instructions except **tlbia**, which is treated as an illegal instruction.

Because the MMU specification for processors that implement the PowerPC architecture is so flexible, it is recommended that the software using these instructions and registers be encapsulated into subroutines to minimize the impact of migrating across the family of implementations.

Table 5-5 summarizes e600 core instructions that specifically control the MMU. For more detailed information about the instructions, refer to Chapter 2, “Registers,” in this book and Chapter 8, “Instruction Set,” in *The Programming Environments Manual*.

Table 5-5. e600 Core Instruction Summary—Control MMUs

Instruction	Description
PowerPC Instructions	
mtsr SR,rS	Move to Segment Register SR[SR#]← rS
mtsrin rS,rB	Move to Segment Register Indirect SR[rB[0–3]]←rS
mfsr rD,SR	Move from Segment Register rD←SR[SR#]
mfsrin rD,rB	Move from Segment Register Indirect rD←SR[rB[0–3]]
PowerPC Optional Instructions	
tlbie rB	TLB Invalidate Entry For effective address specified by rB, TLB[V]←0 The tlbie instruction invalidates all TLB entries indexed by the EA, and operates on both the instruction and data TLBs simultaneously invalidating four TLB entries. The index corresponds to EA[14–19]. In addition, execution of this instruction causes all entries in the congruence class corresponding to the EA to be invalidated in the other processors attached to the same bus. Software must ensure that instruction fetches or memory references to the virtual pages specified by the tlbie instruction have been completed prior to executing the tlbie instruction.
tlbsync	TLB Synchronize Synchronizes the execution of all other tlbie instructions in the system. Specifically, this instruction causes a global (M = 1) TLBSYNC address-only transaction (TT[0–4] = 01001) on the MPX bus. The TLBSYNC transaction terminates normally (without a retry) when all processors on the MPX bus have completed pending TLB invalidations. See Section 5.4.4.2, “TLB Invalidation,” for more detailed information on the tlbsync instruction.
Implementation-Specific Instructions¹	
tlbld	Load Data TLB Entry Loads the contents of the PTEHI and PTELO registers into the DTLB; used for software table searching.
tlbli	Load Instruction TLB Entry Loads the contents of the PTEHI and PTELO registers into the ITLB; used for software table searching.

¹ These instructions are e600-specific.

Table 5-6 summarizes the registers that the operating system uses to program the e600 core MMUs. These registers are accessible to supervisor-level software only with the **mtspr** and **mfspr** instructions. The PowerPC registers are described in Chapter 2, “Register Set,” in *The Programming Environments Manual*. For e600-specific registers, see Chapter 2, “Registers,” of this book.

Table 5-6. e600 Core MMU Registers

Register	Description
PowerPC Registers	
Segment registers (SR0–SR15)	The sixteen 32-bit segment registers are present only in 32-bit implementations of the PowerPC architecture. The fields in the segment register are interpreted differently depending on the value of bit 0. The segment registers are accessed by the mtsr , mtsrin , mfsr , and mfsrin instructions.
BAT registers (IBAT0U–IBAT3U, IBAT0L–IBAT3L, DBAT0U–DBAT3U, and DBAT0L–DBAT3L)	There are 16 BAT registers, organized as four pairs of instruction BAT registers (IBAT0U–IBAT3U paired with IBAT0L–IBAT3L) and four pairs of data BAT registers (DBAT0U–DBAT3U paired with DBAT0L–DBAT3L). These are special-purpose registers that are accessed by the mtspr and mfspr instructions.
SDR1	The SDR1 register specifies the variables used in accessing the page tables in memory. This SPR is accessed by the mtspr and mfspr instructions.
Implementation-Specific Registers	
Only e600-specific: additional BAT registers (IBAT4U–IBAT7U, IBAT4L–IBAT7L, DBAT4U–DBAT7U, and DBAT4L–DBAT7L) ¹	There are 16 additional BAT registers for the e600 core, organized as four pairs of instruction BAT registers (IBAT4U–IBAT7U paired with IBAT4L–IBAT7L) and four pairs of data BAT registers (DBAT4U–DBAT7U paired with DBAT4L–DBAT7L). These SPRs are accessed by the mtspr and mfspr instructions.
SPRG4–SPRG7	The SPRG4–7 provide additional registers to be used by system software for software table searching.
TLBMISS ¹	When software table searching is enabled (HID0[STEN] = 1), and a TLB miss interrupt occurs, the effective address (EA[0–30]) of the instruction or data access that requires the table search is saved in the TLBMISS register.
PTEHI ¹	When software table searching is enabled (HID0[STEN] = 1), and a TLB miss interrupt occurs, the fields of the PTEHI register are loaded automatically with the corresponding SR[VSID] information, and the API of the missed address. The PTEHI register is also used by the tlbli and tlbld instructions.
PTELO ¹	When software table searching is enabled (HID0[STEN] = 1), and a TLB miss interrupt occurs, software determines the lower 32 bits of the PTE and places those bits in the PTELO register. PTELO is also used by the tlbli and tlbld instructions.

¹ These registers are e600-specific.

5.2 Real Addressing Mode

Real addressing is used when either MSR[IR] = 0 or MSR[DR] = 0, and an instruction or data access occurs, respectively. In this case, the default WIMG bits (0b0011) cause data accesses to be considered cacheable (I = 0) and thus load and store accesses are weakly ordered. This is the case even if the data cache is disabled in the HID0 register (as it is out of hard reset). If I/O devices require load and store accesses to occur in strict program order (strongly ordered), translation must be enabled so that the corresponding I bit can be set. Note also, that the G bit must be set to ensure that the accesses are strongly

ordered. For instruction accesses, the default memory access mode bits (WIMG) are also 0b0011. That is, instruction accesses are considered cacheable ($I = 0$), and the memory is guarded. Again, instruction accesses are considered cacheable even if the instruction cache is disabled in the HID0 register (as it is out of hard reset). The W and M bits have no effect on the instruction cache.

For information on the synchronization requirements for changes to MSR[IR] and MSR[DR], refer to [Section 2.3.2.4, “Synchronization,”](#) in this book, and “Synchronization Requirements for Special Registers and for Lookaside Buffers” in Chapter 2, “Register Set,” in the *Programming Environments Manual*.

5.2.1 Real Addressing Mode—32-Bit Addressing

If address translation is disabled ($MSR[IR] = 0$ or $MSR[DR] = 0$) and extended addressing is disabled ($HID0[XAEN] = 0$), for a particular access, the effective address is treated as the 32-bit physical address and is passed directly to the core memory subsystem as described in the “Real Addressing Mode” section in Chapter 7, “Memory Management,” of the *Programming Environments Manual*. In this case only PA[4–35] bit are used and the PA[0–3] bit are cleared.

5.2.2 Real Addressing Mode—Extended Addressing

When address translation is disabled ($MSR[IR] = 0$ or $MSR[DR] = 0$) and extended addressing is enabled ($HID0[XAEN] = 1$), the 36-bit physical address is generated by having the system software add 4 leading zeros to the 32-bit effective address. [Figure 5-6](#) shows how an effective address is converted to a 36-bit physical address for real addressing mode address translation.

5.3 Block Address Translation

The block address translation (BAT) mechanism in the OEA provides a way to map ranges of effective addresses larger than a single page into contiguous areas of physical memory. Such areas can be used for data that is not subject to normal virtual memory handling (paging), such as a memory-mapped display buffer or an extremely large array of numerical data.

Block address translation in the e600 core is described in the “Block Address Translation” section in Chapter 7, “Memory Management,” of the *Programming Environments Manual* for a 32-bit physical address. However, the information that is modified to allow for 36-bit physical addressing is described in the following sections.

The e600 core BAT registers are not initialized by the hardware after the power-up or reset sequence. Consequently, all valid bits in both instruction and data BAT areas must be explicitly cleared before setting any BAT area for the first time and before enabling translation. Also, note that software must avoid overlapping blocks while updating a BAT area or areas. Even if translation is disabled, multiple BAT area hits (with the valid bits set) can corrupt the remaining portion (any bits except the valid bits) of the BAT registers.

Thus multiple BAT hits (with valid bits set) are considered a programming error whether translation is enabled or disabled, and can lead to unpredictable results if translation is enabled, (or if translation is disabled, when translation is eventually enabled). For the case of unused BATs (if translation is to be enabled) it is sufficient precaution to simply clear the valid bits of the unused BAT entries.

5.3.1 BAT Register Implementation of BAT Array—Extended Addressing

The BAT array is comprised of four entries used for instruction accesses and four entries used for data accesses. The BAT array maintains the address translation information for 8 blocks of memory. When using the e600 core, because of the 8 additional BAT registers, the BAT array maintains address translation information for 16 blocks of memory. Each BAT array entry consists of a pair of BAT registers—an upper and a lower BAT register for each entry. The BAT registers are accessed with the **mtspr** and **mfspr** instructions and are only accessible to supervisor-level programs. See Appendix F, “Simplified Mnemonics,” in *The Programming Environments Manual* for a list of simplified mnemonics for use with the BAT registers. The block is defined by a pair of SPRs (upper and lower BAT registers) that contain the effective and physical addresses for the block.

The format and bit definitions of the upper and lower BAT registers for extended addressing are shown in [Figure 5-9](#) and [Figure 5-10](#), respectively.

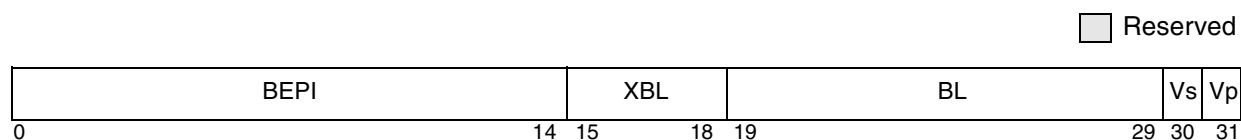
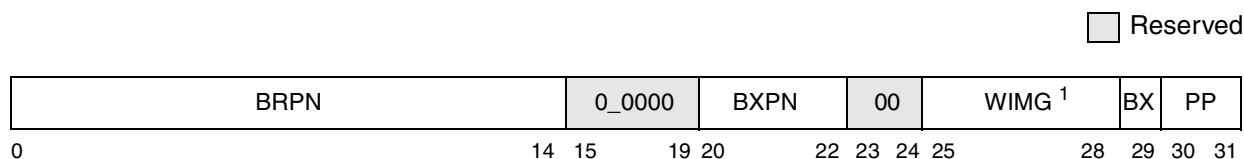


Figure 5-9. Format of Upper BAT Register (BATU)—Extended Block Size



¹ W and G bits are not defined for IBAT registers. Attempting to write to these bits causes boundedly undefined results.

Figure 5-10. Format of Lower BAT Register (BATL)—Extended Addressing

The BAT registers contain the effective-to-physical address mappings for blocks of memory. This mapping information includes the effective address bits that are compared with the effective address of the access, the memory/cache access mode bits (WIMG), and the protection bits for the block. In addition, the size of the block and the starting address of the block are defined by the block physical page number (BRPN) and block size mask (BL) fields.

[Table 5-7](#) describes the bits in the upper and lower BAT registers. Note that the W and G bits are defined for BAT registers that translate data accesses (DBAT registers); attempting to write to the W and G bits in IBAT registers causes boundedly undefined results.

Table 5-7. BAT Registers—Field and Bit Descriptions for Extended Addressing

Upper/Lower BAT ¹	Bits	Name	Description
Upper BAT Register (BAT _{nU})	0–14	BEPI	Block effective page index. This field is compared with high-order bits of the effective address to determine if there is a hit in that BAT array entry.
	15–18	XBL	Extended block length. Used to lengthen block size. 0 When HID0[XBSEN] is cleared at startup, BAT _{nU} [15–18] are always cleared, (0b0000), and extended BAT block size translation does not occur. 1 When HID0[XBSEN] is set at startup, the extended BAT block size is enabled and bits BAT _U [15–18] become the 4 MSBs of the extended 15-bit BL field (BAT _U [15–29]). This allows for extended BAT block sizes of 512 MB, 1 GB, 2 GB, and 4 GB. If HID0[XBSEN] is set at startup and then cleared after startup, the XBL bits will not clear but stay the same as they were set at startup. Values for the extended block length mask are listed in Table 5-9 .
	19–29	BL	Block length. BL is a mask that encodes the size of the block. Values for this field are listed in Table 5-8 .
	30	Vs	Supervisor mode valid bit. This bit interacts with MSR[PR] to determine if there is a match with the effective address. For more information, see the section, “Recognition of Addresses in BAT Arrays,” in <i>The Programming Environments Manual</i> .
	31	Vp	User mode valid bit. This bit also interacts with MSR[PR] to determine if there is a match with the effective address. For more information, see the section, “Recognition of Addresses in BAT Arrays,” in <i>The Programming Environments Manual</i> .
Lower BAT Register (BAT _{nL})	0–14	BRPN	Block physical page number. This field is used in conjunction with the BL field to generate high-order bits of the physical address of the block.
	15–19	—	Reserved
	20–22	BXPN	Block extended physical page number (BXPN). This field comprises bits 0–2 of the physical address.
	23–24	—	Reserved
	25–28	WIMG	Memory/cache access mode bits W Write-through I Caching-inhibited M Memory coherence G Guarded Attempting to write to the W and G bits in IBAT registers causes boundedly-undefined results. For detailed information about the WIMG bits, see Section 3.3.1, “Memory/Cache Access Attributes (WIMG Bits)” .
	29	BX	Block extended physical page number (BX). This field comprises bit 3 of the physical address.
30–31	PP	Protection bits for block. This field determines the protection for the block as described in the section, “Block Memory Protection,” in <i>The Programming Environments Manual</i> .	

¹ A context synchronizing instruction must follow an mtspr.

BAT_n registers can be accessed with **mtspr** and **mfspr**. For synchronization requirements on the BAT_n registers see [Table 2-37](#).

The BL field in the upper BAT register is a mask that encodes the size of the block. [Table 5-8](#) defines the bit encoding for the BL field of the upper BAT register.

Table 5-8. Upper BAT Register Block Size Mask Encoding

Block Size	BATU[BL] Encoding
128 Kbytes	000 0000 0000
256 Kbytes	000 0000 0001
512 Kbytes	000 0000 0011
1 Mbyte	000 0000 0111
2 Mbytes	000 0000 1111
4 Mbytes	000 0001 1111
8 Mbytes	000 0011 1111
16 Mbytes	000 0111 1111
32 Mbytes	000 1111 1111
64 Mbytes	001 1111 1111
128 Mbytes	011 1111 1111
256 Mbytes	111 1111 1111

Only the values shown in [Table 5-8](#) are valid for BL. An effective address is determined to be within a BAT area if the appropriate bits (determined by the BL field) of the effective address match the value in the BEPI field of the upper BAT register and if the appropriate valid bit (Vs or Vp) is set. Note that for an access to occur, the protection bits (PP bits) in the lower BAT register must be set appropriately, as described and defined in Chapter 7, “Memory Management,” in *The Programming Environments Manual*.

The number of zeros in the BL field determines the bits of the effective address that are used in the comparison with the BEPI field to determine if there is a hit in that BAT array entry. The right-most bit of the BL field is aligned with bit 14 of the effective address; bits of the effective address corresponding to ones in the BL field are then cleared to zero for the comparison.

The value loaded into the BL field determines both the size of the block and the alignment of the block in both effective address space and physical address space. The values loaded into the BEPI and BRPN fields must have at least as many low-order zeros as there are ones in BL. Otherwise, the results are undefined.

5.3.2 Block Physical Address Generation—Extended Addressing

When extended addressing is enabled ($HID0[XAEN] = 1$) and the block protection mechanism validates the access, then a 36-bit physical address is formed as shown in [Figure 5-11](#). Bits in the effective address corresponding to ones in the BL field, concatenate with the 17 low-order bits of the effective address, and form the offset within the block of memory defined by the BAT array entry. Bits in the effective address corresponding to zeros in the BL field are then logically ORed with the corresponding bits in the BRPN field to form the next high-order bits of the physical address. The highest-order four bits of the BRPN field (BATL[0–3]) form bits 4–7 of the physical address (PA[4–7]). Finally, the four extended address bits from

BATL[BXPN] and BATL[BX] are concatenated to form the highest-order four bits of the physical address (PA[0–2] and PA[3], respectively).

Figure 5-11 shows how a block physical address is generated for extended addressing.

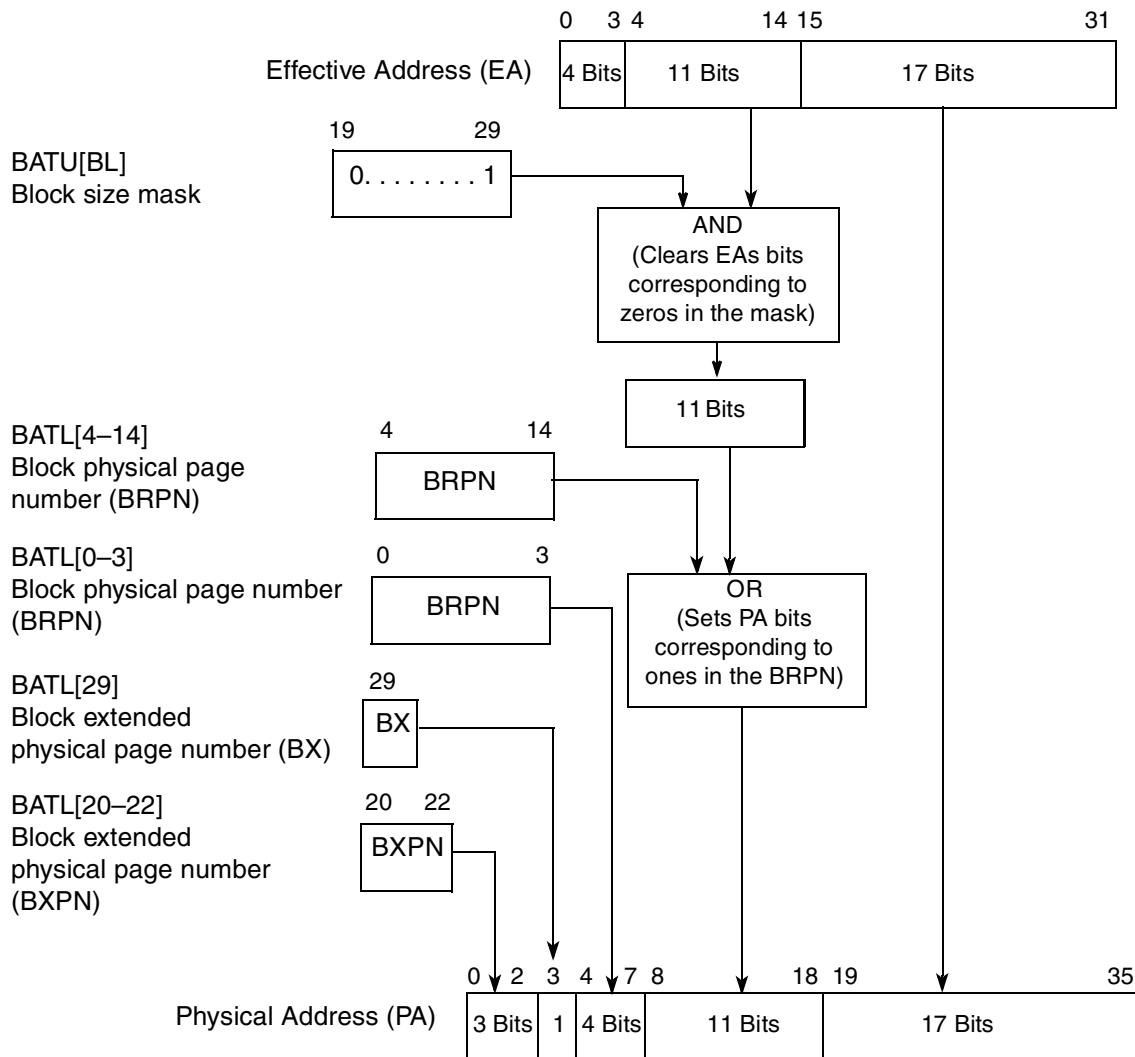


Figure 5-11. Block Physical Address Generation—Extended Addressing

5.3.2.1 Block Physical Address Generation with an Extended BAT Block Size

When the extended BAT block size is enabled (HID0[XBSEN]=1) the BAT block size is increased through the XBL field in the Upper BAT register, as shown in Figure 5-9. This allows for extended BAT block sizes of 512MB, 1 GB, 2GB, and 4 GB. If HID0[XBSEN] is set at startup and then cleared after startup, the XBL bits do not clear but stay the same as they were set at startup. The BL field is extended to 15 bits, with the XBL bits becoming the 4 most-significant bits (MSBs) for the block size. The encoding for the extended BL field is shown in Table 5-9.

Table 5-9. Upper BAT Register Block Size Mask Encoding when the Extended Block Size is Enabled (HID0[XBSEN] = 1)

Block Size	BATU[XBL + BL] Encoding
128 Kbytes	000 0000 0000 0000
256 Kbytes	000 0000 0000 0001
512 Kbytes	000 0000 0000 0011
1 Mbyte	000 0000 0000 0111
2 Mbytes	000 0000 0000 1111
4 Mbytes	000 0000 0001 1111
8 Mbytes	000 0000 0011 1111
16 Mbytes	000 0000 0111 1111
32 Mbytes	000 0000 1111 1111
64 Mbytes	000 0001 1111 1111
128 Mbytes	000 0011 1111 1111
256 Mbytes	000 0111 1111 1111
512 Mbytes	000 1111 1111 1111
1 Gbytes	001 1111 1111 1111
2 Gbytes	011 1111 1111 1111
4 Gbytes	111 1111 1111 1111

Only the values shown in [Table 5-9](#) are valid for an extended 15-bit BL field. An effective address is determined to be within a BAT area if the appropriate bits (determined by the XBL and BL fields) of the effective address match the value in the 15-bit BEPI field of the upper BAT register and if the appropriate valid bit (Vs or Vp) is set.

The number of zeros in the extended BL field determines the bits of the effective address that are used in the comparison with the BEPI field to determine if there is a hit in that BAT array entry. The right-most bit of the BL field is still aligned with bit 14 of the effective address; bits of the effective address corresponding to ones in the BL field are then cleared to zero for the comparison.

The value loaded into the BL field determines both the size of the block and the alignment of the block in both effective address space and physical address space. The values loaded into the BEPI and BRPN fields must have at least as many low-order zeros as there are ones in BL, otherwise the results are undefined.

[Figure 5-12](#) shows how a block physical address is generated for an extended block size with extended addressing (36-bit physical address).

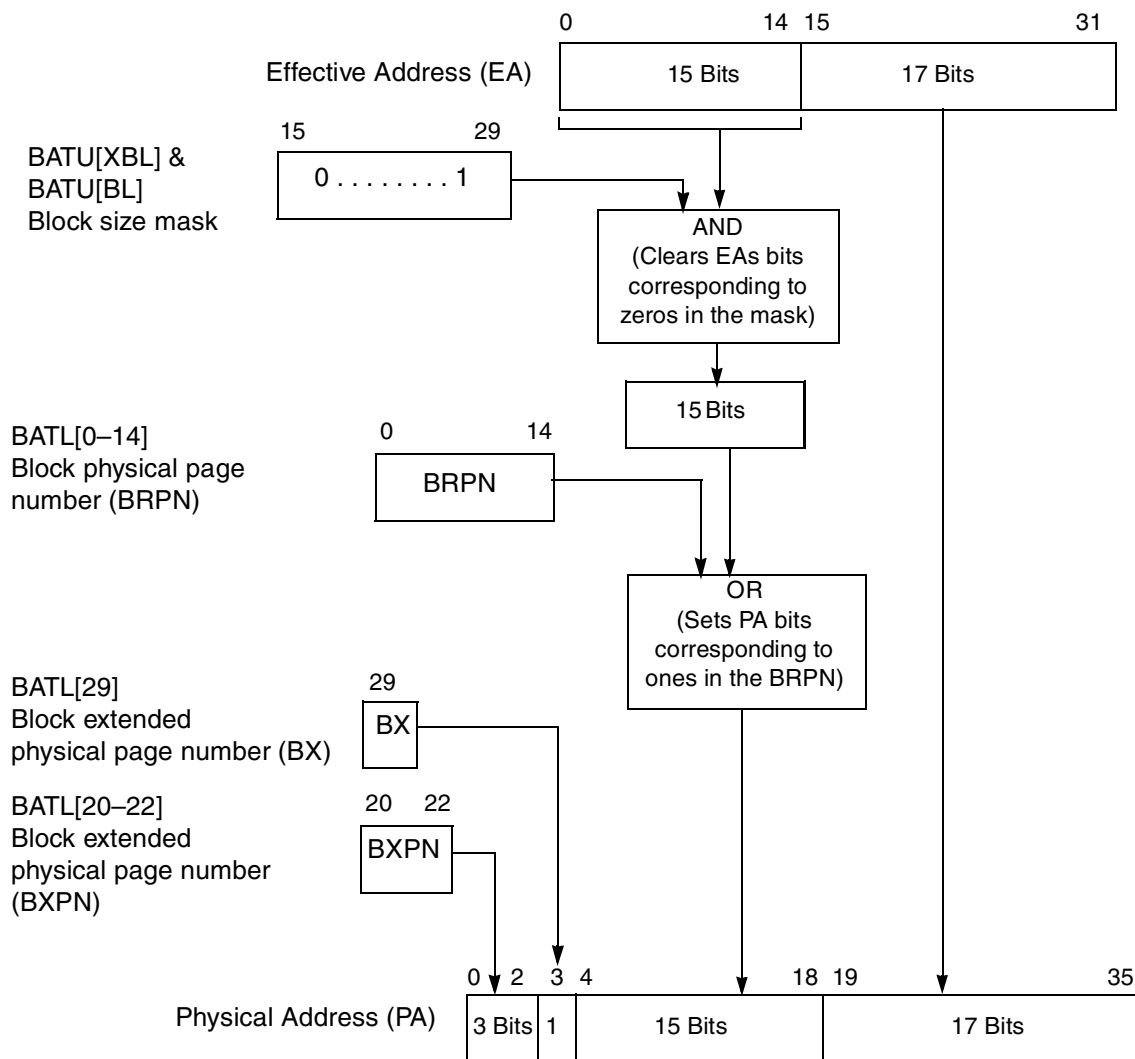


Figure 5-12. Block Physical Address Generation—Extended Block Size for a 36-Bit Physical Address

5.3.3 Block Address Translation Summary—Extended Addressing

Figure 5-13 is an expansion of the ‘BAT Array Hit’ branch of Figure 5-7 and shows the translation of address bits when extended addressing is enabled ($HID0[XAEN] = 1$) so that a 36-bit physical address is generated. Extended address bits from the lower BAT register are concatenated to the highest order bits of the physical address. Note that the figure does not show when many of the interrupts in Table 5-3 are detected or taken as this is implementation-specific. For further details on memory protection violations see the section, “Block Memory Protection,” of *The Programming Environments Manual*.

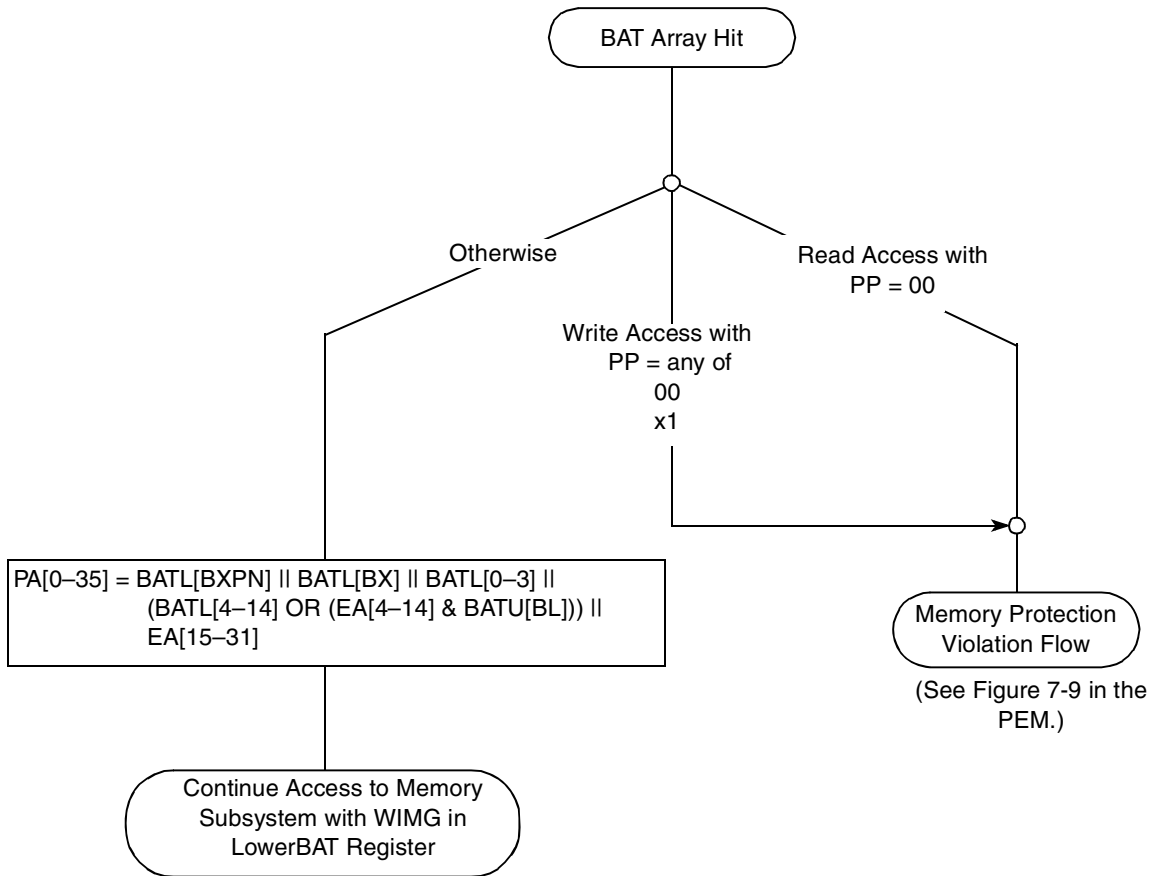


Figure 5-13. Block Address Translation Flow—Extended Addressing

Figure 5-14 shows translation of address bits when the extended block size is enabled (HID0[XBSEN]=1) and extended addressing is enabled (HID0[XAEN] = 1). (Note that extended addressing does not have to be turned on to extend the BAT size.) In this case all 15 bits of the effective address are compared with the BEPI field to determine if there is a hit in the BAT array. Once a match has been found, the physical address is generated by using all the bits in the effective address that correspond to zeros in the BL field. The result is then logically ORed with the BRPN field to form bits 4–18 of the physical address.

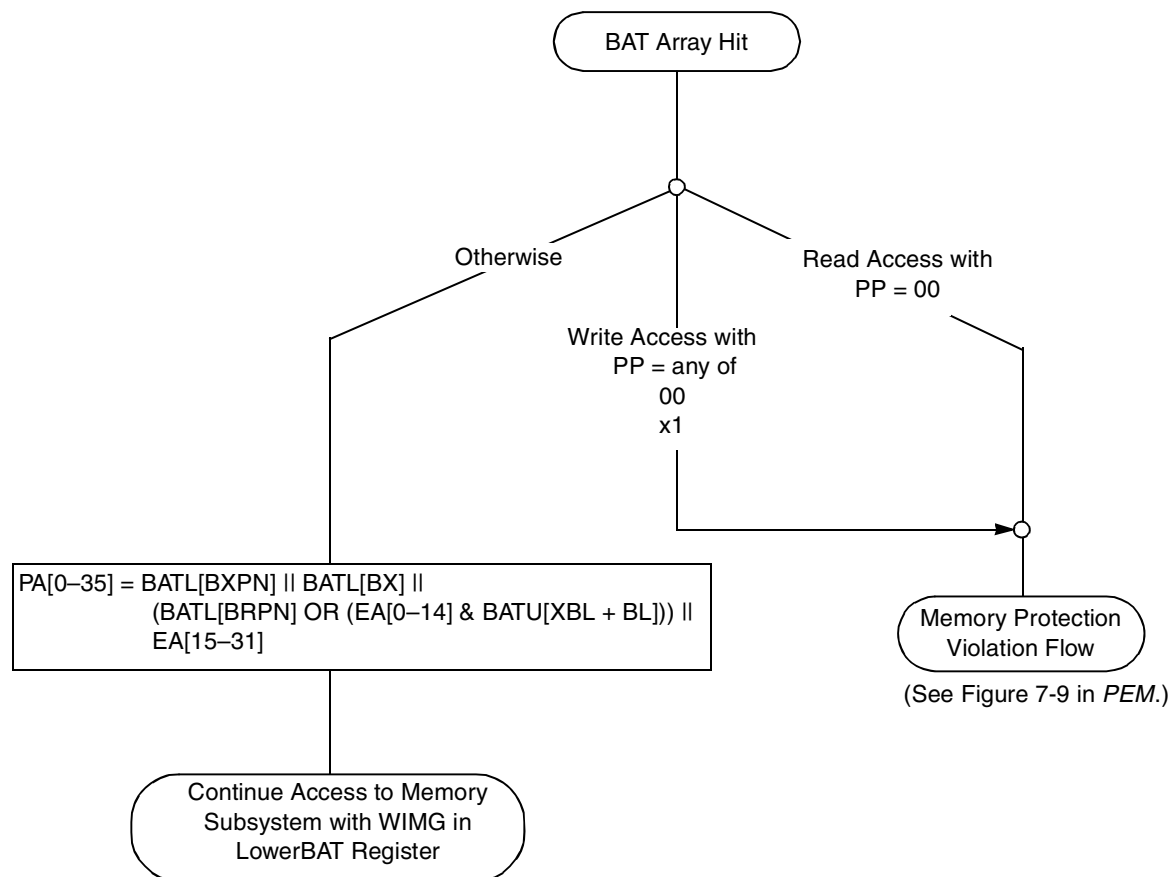


Figure 5-14. Block Address Translation Flow—Extended Block Size for a 36-Bit Physical Address

5.4 Memory Segment Model

The e600 core adheres to the memory segment model as defined in Chapter 7, “Memory Management,” in *The Programming Environments Manual* for 32-bit implementations. Memory in the PowerPC OEA is divided into 256-Mbyte segments. This segmented memory model provides a way to map 4-Kbyte pages of effective addresses to 4-Kbyte pages in physical memory (page address translation), while providing the programming flexibility afforded by a large virtual address space (52 bits).

The segment/page address translation mechanism may be superseded by the block address translation (BAT) mechanism described in Section 5.3, “Block Address Translation.” If there is not a BAT hit, the page address translation proceeds in the following two steps:

1. From effective address to the virtual address (that never exists as a specific entity but can be considered to be the concatenation of the virtual page number and the byte offset within a page)
2. From virtual address to physical address

The following subsections highlight those areas of the memory segment model defined by the OEA that are specific to the e600 core as well as modifications that apply for extended 36-bit physical addressing. The memory segment model for 32-bit physical addressing is as described in Chapter 7, “Memory Management,” in *The Programming Environments Manual*.

5.4.1 Page Address Translation Overview

A page address translation overview for 32-bit physical addresses is provided in the section, “Page Address Translation Overview,” of *The Programming Environments Manual*. The following sections highlight the differences for 36-bit physical addressing. The general flow for page address translation is as shown in [Figure 5-15](#). The effective address, EA[0–3], is used to find the correct segment descriptor in the segment registers. The segment descriptor is then used to generate the 52-bit virtual address (VA). The MMU then fetches the page table entry (PTE) from the virtual address. If the PTE is not found in the tables then a hardware or software page table search is performed. The following subsections describe the details of how page address translation is performed for an extended 36-bit physical address.

The translation of an effective address to an extended physical address is shown in [Figure 5-15](#). Note that in the process of translating the physical address, a 52-bit virtual address is generated and that is used to find the PTE in the TLB or through a hardware or software table search operation. The physical address translation is as follows:

- Bits 0–3 of the effective address comprise the segment register number used to select a segment descriptor, from which the virtual segment ID (VSID) is extracted.
- Bits 4–19 of the effective address bits correspond to the page number within the segment. EA[4–9] define the abbreviated page index (API), and EA[10–13] define the extended API (EAPI) bits in the PTE. EA[4–19] are concatenated with the VSID from the segment descriptor to form the virtual page number (VPN). The VPN is used to search for the PTE in either a TLB within the core or the page table. The PTE then provides the physical page number (RPN) and the extended page number bits (XPN and X). The XPN and X fields of the page table entry (PTE) provide the extra bits for the extended physical page number. These become the most significant bits of the 36-bit physical address (PA[0–3]).
- Bits 20–31 of the effective address are the byte offset within the page; these are bits 24–35 of the physical address used to access memory.

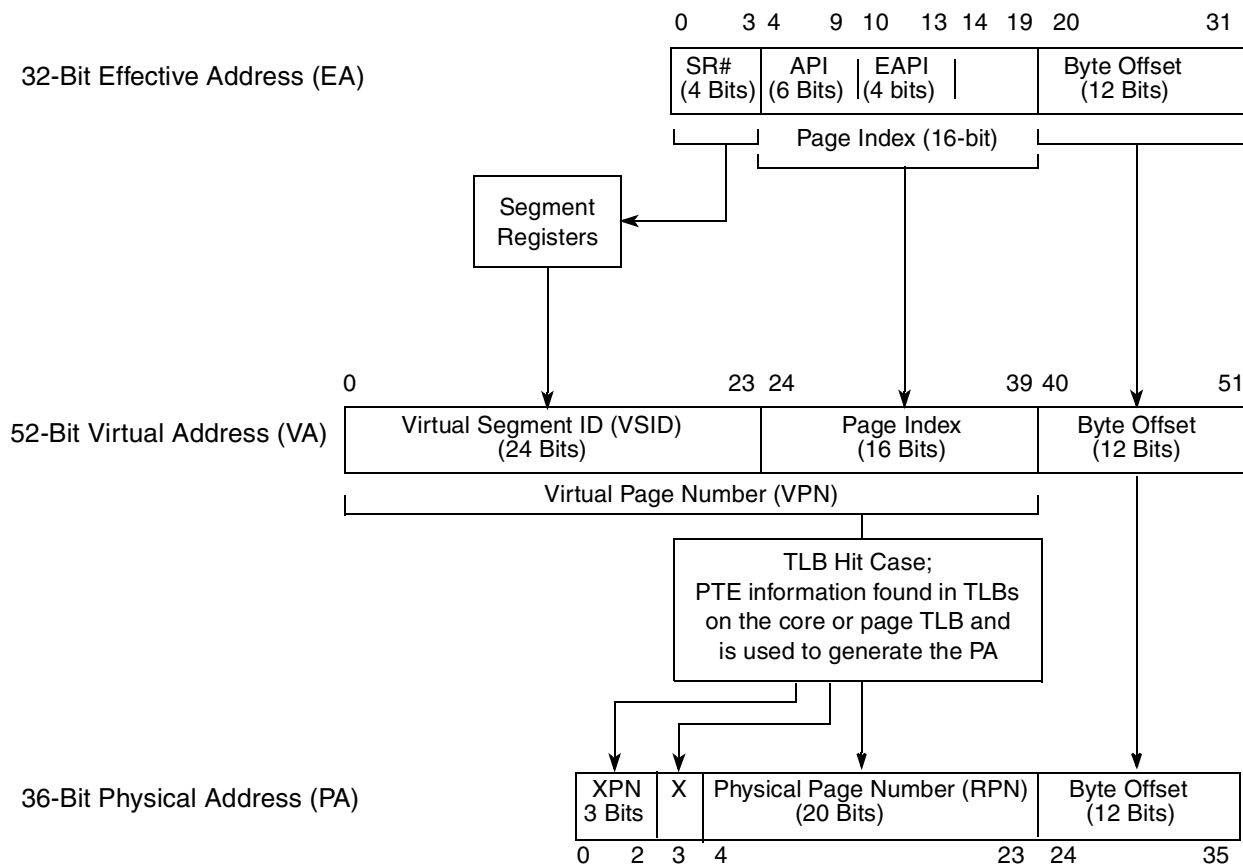


Figure 5-15. Generation of Extended 36-Bit Physical Address for Page Address Translation

5.4.1.1 Segment Descriptor Definitions

The segment registers are defined the same for both 32- and 36-bit physical addressing. See the description of the segment register format in the “Segment Descriptor Format” section of Chapter 7, “Memory Management,” in *The Programming Environments Manual*. The segment descriptors are 32 bits long and reside in one of the 16 segment registers within the core. The fields in the segment register are interpreted differently depending on the value of the T bit. When T=1 (SR_n[T] = 1), the segment descriptor defines a direct-store segment; however, the e600 core does not support the direct-store interface. When an access is determined to be to the direct-store interface space, the core takes a DSI interrupt if it is a data access (see Section 4.6.3, “DSI Interrupt (0x00300)”), and takes an ISI interrupt if it is an instruction access (see Section 4.6.4, “ISI Interrupt (0x00400)”).

5.4.2 Page History Recording

Referenced (R) and changed (C) bits in each PTE keep history information about the page. When hardware table searching is enabled, the history bits are maintained by a combination of the e600 core table search hardware and the system software. When software table searching is enabled, the history bits are maintained by a combination of the following:

- Table search software provided by the interrupt
- Interrupt model

The operating system uses the information in each PTE to determine which areas of memory to write back to disk when new pages must be allocated in main memory

Referenced and changed recording is performed only for accesses made with page address translation and not for translations made with the BAT mechanism or for accesses that correspond to direct-store ($T = 1$) segments. Furthermore, R and C bits are maintained only for accesses made while address translation is enabled ($MSR[IR] = 1$ or $MSR[DR] = 1$).

While these bits are initially programmed by the operating system into the page table, the architecture specifies that the R and C bits may be maintained either by the processor hardware (automatically) or by some software-assist mechanism that updates these bits when required. Software table searching is optional in the e600 core. When software table searching is enabled ($HID0[STEN] = 1$), the software table search routines are responsible for setting the R bit when a PTE is accessed. Additionally, the e600 core also causes an interrupt (to vector to the software table search routines) when the C bit in the corresponding TLB entry (and PTE entry) requires updating.

In the e600 core, the referenced and changed bits are updated as follows:

- For TLB hits, the C bit is updated according to [Table 5-11](#).
- For TLB misses, when a table search operation is in progress to locate a PTE. The R and C bits are updated (set, if required) to reflect the status of the page based on this access.

Table 5-11. Table Search Operations to Update History Bits—TLB Hit Case

R and C bits in TLB Entry	Processor Action
00	Combination does not occur
01	Combination does not occur
10	Read: No special action Write: Table search operation required to update C. Causes a data TLB miss on store interrupt.
11	No special action for read or write

[Table 5-11](#) shows that the status of the C bit in the TLB entry (in the case of a TLB hit) is what causes the processor to update the C bit in the PTE (the R bit is assumed to be set in the page tables if there is a TLB hit). Therefore, when software clears the R and C bits in the page tables in memory, it must invalidate the TLB entries associated with the pages whose referenced and changed bits were cleared.

In some previous implementations, the **dcbt** and **dcbtst** instructions execute only if there is a TLB/BAT hit or if the processor is in real addressing mode. In case of a TLB or BAT miss, these instructions are

treated as no-ops and do not initiate a table search operation, and do not set either the R or C bits. In the e600 core, the **dcbt**, **dcbtst**, and data stream touch instructions (**dst[t]** and **dstst[t]**) do cause a table search operation in the case of a TLB miss. However, they never cause the C bit to be set, and a failed table search operation does not cause an interrupt.

As defined by the PowerPC architecture, the referenced and changed bits are updated as if address translation were disabled (real addressing mode). If these update accesses hit any of the caches within the core, they are not seen on the MPX bus. If they miss in the caches, they are performed as typical cache line fill accesses on the MPX bus (if the data cache is enabled), or as discrete read and write accesses (if the data cache is disabled).

5.4.2.1 Referenced Bit

The referenced (R) bit of a page is located in the PTE in the page table. Every time a page is referenced (with a read or write access) and the R bit is zero, the R bit is set in the page table. The OEA specifies that the referenced bit may be set immediately, or the setting may be delayed until the memory access is determined to be successful. Because the reference to a page is what causes a PTE to be loaded into the TLB, the referenced bit in all e600 core TLB entries is effectively always set. The processor never automatically clears the referenced bit.

The referenced bit is only a hint to the operating system about the activity of a page. At times, the referenced bit may be set although the access was not logically required by the program or even if the access was prevented by memory protection. Examples of this in systems include the following:

- Fetching of instructions not subsequently executed
- A memory reference caused by a speculatively executed instruction that is mispredicted
- Accesses generated by an **lswx** or **stswx** instruction with a zero length
- Accesses generated by an **stwcx.** instruction when no store is performed because a reservation does not exist
- Accesses that cause interrupts and are not completed

5.4.2.2 Changed Bit

The changed bit of a page is located both in the PTE in the page table and in the copy of the PTE loaded into the TLB (if a TLB is implemented, as in the e600 core). Whenever a data store instruction is executed successfully, if the TLB search (for page address translation) results in a hit, the changed bit in the matching TLB entry is checked. If the C bit is already set, it is not updated. If the TLB changed bit is 0, the e600 core initiates a table search operation to set the C bit in the corresponding PTE in the page table. The e600 core then reloads the TLB (with the C bit set). This occurs automatically when hardware table searching is enabled. When software table searching is enabled, the e600 core takes a data TLB miss on store interrupt for this case so that the software can perform the table search operation to set the C bit. Refer to [Section 5.5.5, “Implementation-Specific Software Table Search Operation,”](#) for an example code sequence that handles these conditions.

The changed bit (in both the TLB and the PTE in the page tables) is set only when a store operation is allowed by the page memory protection mechanism and the store is guaranteed to be in the execution path

(unless an interrupt, other than those caused by the **sc**, **rfi**, or trap instructions, occurs). Furthermore, the following conditions may cause the C bit to be set:

- The execution of an **stwcx.** instruction is allowed by the memory protection mechanism but a store operation is not performed.
- The execution of an **stswx** instruction is allowed by the memory protection mechanism but a store operation is not performed because the specified length is zero.
- The store operation is not performed because an interrupt occurs before the store is performed.

Again, note that the execution of the **dcbt**, **dcbtst** and data stream touch instructions (**dst[t]** and **dstst[t]**) never cause the C bit to be set.

5.4.2.3 Scenarios for Referenced and Changed Bit Recording

This section provides a summary of the model (defined by the PowerPC OEA) that is used by processors for maintaining the referenced and changed bits. In some scenarios, the bits are guaranteed to be set by the processor; in some scenarios, the architecture allows that the bits may be set (not absolutely required), and in some scenarios, the bits are guaranteed to not be set. Note that when the e600 core updates the R and C bits in memory, the accesses are performed as if $MSR[DR] = 0$ and $G = 0$ (that is, as nonguarded cacheable operations in which coherency is required— $WIMG = 0010$).

When software table searching is enabled, the e600 core does not maintain the R and C bits in hardware, and software assistance is required. In this case, the information in this section still applies, except that the software performing the updates is constrained to the rules described (that is, the software must set bits shown as guaranteed to be set and not set bits shown as guaranteed not to be set).

[Table 5-12](#) defines a prioritized list of the R and C bit settings for all scenarios. The entries in the table are prioritized from top to bottom, such that a matching scenario occurring closer to the top of the table takes precedence over a matching scenario closer to the bottom of the table. For example, if an **stwcx.** instruction causes a protection violation and there is no reservation, the C bit is not altered, as shown for the protection violation case. Note that in the table, load operations include those generated by load instructions and by the cache management instructions that are treated as a load with respect to address translation. Similarly, store operations include those operations generated by store instructions and by the cache management instructions that are treated as a store with respect to address translation.

In the columns for the e600 core, the combination of the e600 core itself and the software used to search the page tables described in [Section 5.5.5, “Implementation-Specific Software Table Search Operation,”](#) is assumed. For more information, see “Page History Recording” of *The Programming Environments Manual*.

Table 5-12. Model for Guaranteed R and C Bit Settings

Priority	Scenario	Causes Setting of R Bit		Causes Setting of C Bit	
		OEA	e600 Core	OEA	e600 Core
1	No-execute protection violation	No	No	No	No
2	Page protection violation	Maybe	Yes	No	No

Table 5-12. Model for Guaranteed R and C Bit Settings (continued)

Priority	Scenario	Causes Setting of R Bit		Causes Setting of C Bit	
		OEA	e600 Core	OEA	e600 Core
3	Out-of-order instruction fetch or load operation	Maybe	No	No	No
4	Out-of-order store operation. Would be required by the sequential execution model in the absence of system-caused or imprecise interrupts, or of floating-point assist interrupt for instructions that would cause no other kind of precise interrupt.	Maybe ¹	No	No	No
5	All other out-of-order store operations	Maybe ¹	No	Maybe ¹	No
6	Zero-length load (lswx)	Maybe	No	No	No
7	Zero-length store (stswx)	Maybe ¹	No	Maybe ¹	No
7.5	Store that triggers a precise interrupt (DSI, ALI)	Maybe	Yes	Maybe	Maybe
8	Store conditional (stwcx.) that does not store	Maybe ¹	Yes	Maybe ¹	Yes
9	In-order instruction fetch	Yes ²	Yes	No	No
10	Load instruction	Yes	Yes	No	No
11	Store instruction or dcbz instruction	Yes	Yes	Yes	Yes
12	icbi , dcbt , or dcbtst instruction	Maybe	No	No	No
13	dcbst or dcbf instruction	Maybe	Yes	No	No
14	dcbi instruction	Maybe ¹	Yes	Maybe ¹	Yes
15	dst instruction	n/a	Yes	n/a	No

¹ If C is set, R is guaranteed to be set also.

² Includes the case in which the instruction is fetched out of order and R is not set (does not apply for e600 core).

5.4.3 Page Memory Protection

The e600 core implements page memory protection as it is defined in the section, “Page Memory Protection,” of *The Programming Environments Manual*.

5.4.4 TLB Description

The e600 core implements separate 128-entry data and instruction TLBs to maximize performance. This section describes the hardware resources provided in the e600 core to facilitate page address translation. Note that the hardware implementation of the MMU is not specified by the architecture, and while this description applies to the e600 core, it does not necessarily apply to other processors that implement the PowerPC architecture.

5.4.4.1 TLB Organization and Operation

Because the e600 core has two MMUs (IMMU and DMMU) that operate in parallel, some of the MMU resources are shared, and some are actually duplicated (shadowed) in each MMU to maximize performance. For example, although the architecture defines a single set of segment registers for the

MMU, the core maintains two identical sets of segment registers, one for the IMMU and one for the DMMU; when an instruction that updates the segment register executes, the core automatically updates both sets.

The TLB entries contain copies of PTEs in the page tables in memory and are similar in structure. To uniquely identify a TLB entry as the required PTE, the TLB entry also contains four more bits of the page index, EA[10–13], called the extended API (EAPI) in addition to the API bits in the PTE.

Each TLB contains 128 entries organized as a two-way set-associative array with 64 sets as shown in Figure 5-17 for the DTLB (the ITLB organization is the same). When an address is being translated, a set of two TLB entries is indexed in parallel with the access to a segment register. If the address in one of the two TLB entries is valid and matches the 40-bit virtual page number, that TLB entry contains the translation. If no match is found, a TLB miss occurs.

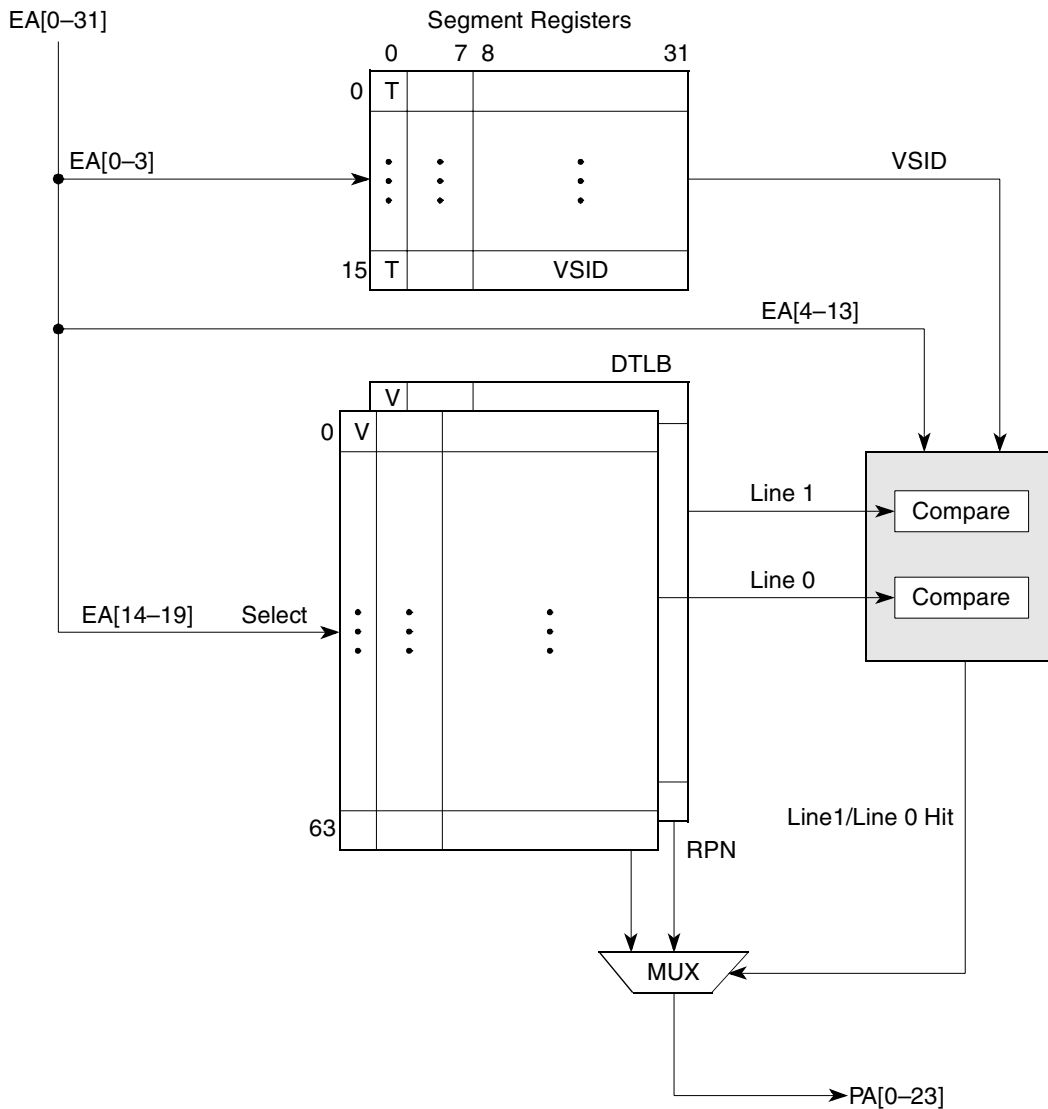


Figure 5-17. Segment Register and DTLB Organization

Unless the access is the result of an out-of-order access, when $HID0[STEN] = 0$, a hardware table search operation begins if there is a TLB miss. If the access is out of order, the table search operation is postponed until the access is required; at that point the access is no longer out of order. When the matching PTE is found in memory, it is loaded into the TLB entry selected by the least-recently-used (LRU) replacement algorithm, and the translation process begins again, this time with a TLB hit.

A software table search is initiated when $HID0[STEN] = 1$ and a TLB miss occurs. In this case, e600 core causes an interrupt when the TLB and BAT both miss for an access. There are separate interrupt vectors for instruction fetches, data loads, and data stores. Refer to [Section 5.5.5, “Implementation-Specific Software Table Search Operation,”](#) for more information on the loading of the TLBs in this case.

Each set of TLB entries has one associated LRU bit. The LRU bit for a set is updated any time either entry is used, even if the access is speculative. Invalid entries are always the first to be replaced.

Although both MMUs can be accessed simultaneously (both sets of segment registers and TLBs can be accessed in the same clock), only one exception condition is reported at a time. Interrupts are processed in strict program order, and a particular interrupt is processed when the instruction that caused it is the next instruction to be retired. When a particular instruction causes an instruction MMU interrupt, that interrupt is processed before that instruction can cause a data MMU interrupt.

ITLB miss conditions are reported when there are no more instructions to be dispatched or retired (the pipeline is empty), and DTLB miss conditions are reported when the load or store instruction is the next instruction to be retired. In the case that both an ITLB and DTLB miss are reported in the same clock, the DTLB miss takes precedence and is handled first. Refer to [Chapter 6, “Instruction Timing,”](#) for more detailed information about the internal pipelines and the reporting of interrupts.

Although address translation is disabled on a soft or hard reset condition, the valid bits of TLB entries are not automatically cleared. Thus TLB entries must be explicitly cleared by the system software (with a series of **tlbie** instructions) before address translation is enabled. Also, note that the segment registers do not have a valid bit, and so they should also be initialized before translation is enabled.

5.4.4.2 TLB Invalidation

The e600 core implements the optional **tlbie** and **tlbsync** instructions, that are used to invalidate TLB entries.

The **tlbia** instruction is not implemented on the e600 core and when its opcode is encountered, an illegal instruction program interrupt is generated. To invalidate all entries of both TLBs, 64 **tlbie** instructions must be executed, incrementing the value in $EA[14-19]$ by one each time. See Chapter 8, “Instruction Set,” in *The Programming Environments Manual* for architecture information about the **tlbie** instruction.

5.4.4.2.1 **tlbie** Instruction

The execution of the **tlbie** instruction always invalidates four entries—for each ITLB set and each DTLB set, one entry in each of the two ways is indexed by $EA[14-19]$. The **tlbie** instruction executes regardless of the setting of the $MSR[DR]$ and $MSR[IR]$ bits.

The architecture allows **tlbie** to optionally enable a TLB invalidate signaling mechanism in hardware so that other processors also invalidate their resident copies of the matching PTE. When an e600 core

TLBIQ empties. If the TLBIQ is invalid and the transaction is not retried by any other processor, the e600 core loads the TLBIQ with EA[14–19] and sets the TLBIQ valid bit. This causes the e600 core to invalidate the four TLB entries (both the ITLB and DTLB entries indexed by EA[14–19]), and internally mark all accesses with previously translated addresses.

The **tlbie** instruction does not affect the instruction fetch operation—that is, the prefetch buffer is not purged and the machine does not cause these instructions to be refetched.

5.4.4.2.2 **tlbsync** Instruction

The **tlbsync** instruction ensures that all previous **tlbie** instructions executed by the system have completed. Specifically, **tlbsync** causes a global ($M = 1$) TLBSYNC address-only transaction ($TT[0–4] = 01001$) on the MPX bus if that processor has completed all previous **tlbie** instructions and any memory operations based on the contents of those invalidated TLB entries have propagated through to completion.

Execution of a **tlbsync** instruction affects outstanding VTQ operations in the same way as a **sync** instruction, (see [Chapter 7, “AltiVec Technology Implementation”](#)) with the following additional effect: an outstanding table search operation for a VTQ-initiated access is cancelled when **tlbsync** is dispatched to the LSU, possibly causing a line fetch skip as described in [Section 5.5.2, “Page Table Search Operations—Implementation.”](#)

The **tlbsync** instruction does not complete until it is the oldest instruction presented to the memory subsystem on the core. This occurs when all of the following conditions exist:

- The **tlbsync** instruction is the oldest instruction in the store queue
- The instruction and data cache reload tables are idle
- There are no outstanding table search operations (note that a table search operation for a VTQ-initiated access may have been cancelled as described above)

[Figure 5-19](#) shows the flow of events caused by execution of the **tlbsync** instruction as well as the actions taken by the e600 core when a TLBSYNC transaction is detected on the MPX bus.

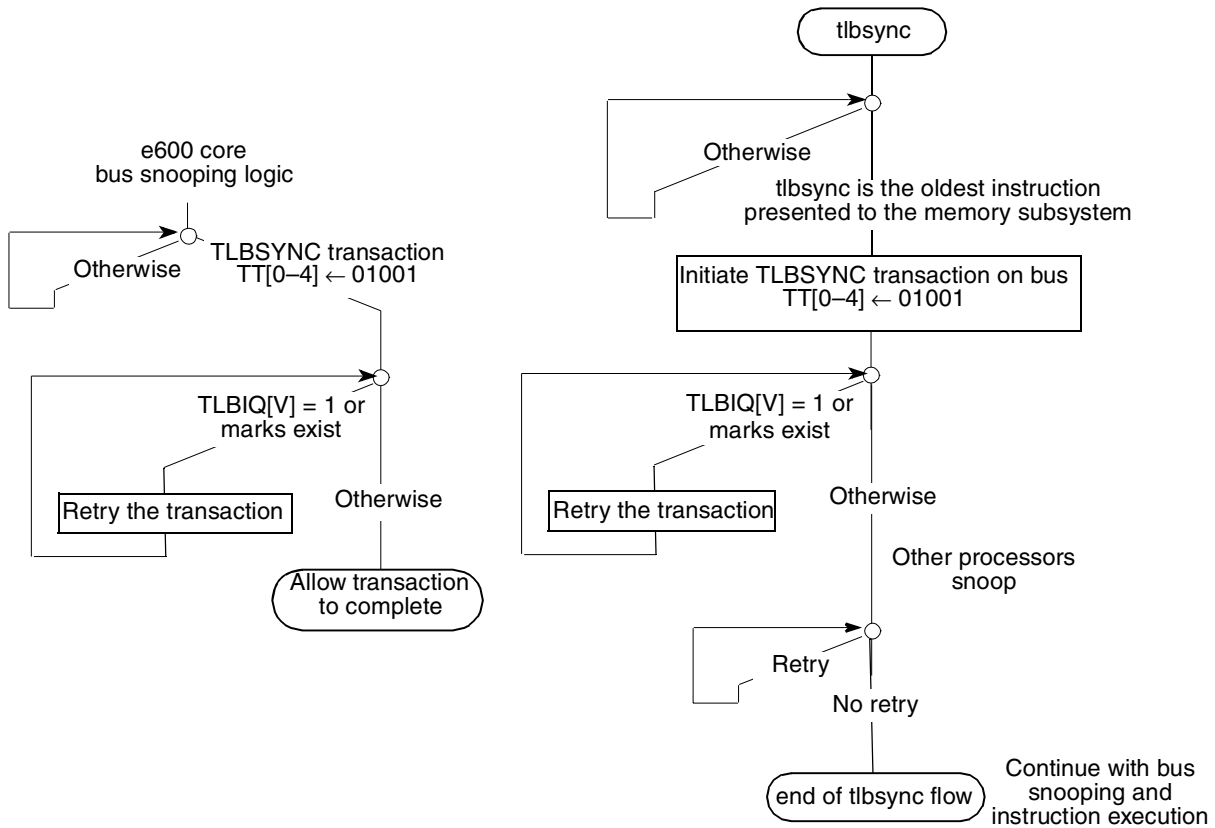


Figure 5-19. tbsync Instruction Execution and MPX Bus Snooping Flow

When an e600 core processor detects a TLBSYNC broadcast transaction, it causes a retry of that transaction until all pending TLB invalidate operations have completed. In this snoop process, the core checks its TLBIQ and any pending marks for previously translated addresses. If the queue is valid or if any marks exist, the TLBSYNC transaction is retried, until the queue is invalid (idle) and no marks exist.

5.4.4.2.3 Synchronization Requirements for tlbie and tbsync

In order to guarantee that a particular processor with an e600 core executing a **tlbie** instruction has completed the operation, a **sync** instruction must be placed after the **tlbie** instruction. A **tbsync** instruction can also be used instead of the **sync** instruction for this purpose, but a **sync** will suffice for that processor. However, in order to guarantee that all processors containing e600 cores in a system have coherently invalidated their respective TLB entries due to a **tlbie** instruction executing on any one of those processors, a **tbsync** instruction is required.

The PowerPC architecture requires that when a **tbsync** instruction has been executed by a processor, a **sync** instruction must be executed by that processor before a **tlbie** or **tbsync** instruction is executed by another processor. If this requirement is not met, a livelock situation may occur in a system with multiple processors with e600 cores. Specifically, if more than one processor executes **tlbie** or **tbsync** instructions simultaneously, it is likely that these processors will cause a system livelock.

5.4.5 Page Address Translation Summary—Extended Addressing

A detailed description of page address translation for a 32-bit physical address is provided in the section, “Page Address Translation Summary,” of Chapter 7, “Memory Management,” in the *Programming Environments Manual*. The following section highlights the differences for 36-bit physical addressing.

[Figure 5-20](#) provides the detailed flow for the page address translation mechanism when using extended addressing.

When an instruction or data access occurs, the effective address is routed to the appropriate MMU. EA0–EA3 select 1 of the 16 segment registers, and the remaining effective address bits and the VSID field from the segment register are passed to the TLB. EA[14–19] then select two entries in the TLB; the valid bits are checked, and the 40-bit virtual page number (24-bit VSID concatenated with EA[4–19]) must match the VSID, EAPI, and API fields of the TLB entries. If one of the entries hits, the PP bits are checked for a protection violation. If these bits do not cause an interrupt, the C bit is checked. If the C bit must be updated, a table search operation is initiated. If the C bit does not require updating, the RPN value with the XPN and X extensions is passed to the core memory subsystem and the WIMG bits are then used as attributes for the access.

[Figure 5-20](#) includes the checking of the N bit in the segment descriptor and then expands on the ‘TLB Hit’ branch of [Figure 5-8](#). The detailed flow for the ‘TLB Miss’ branch of [Figure 5-8](#) is described in [Section 5.5.2, “Page Table Search Operations—Implementation.”](#) Note that as in the case of block address translation, if an attempt is made to execute a **dcbz** instruction to a page marked either write-through or caching-inhibited ($W = 1$ or $I = 1$), an alignment interrupt is generated. The checking of memory protection violation conditions is described in Chapter 7, “Memory Management,” in *The Programming Environments Manual*.

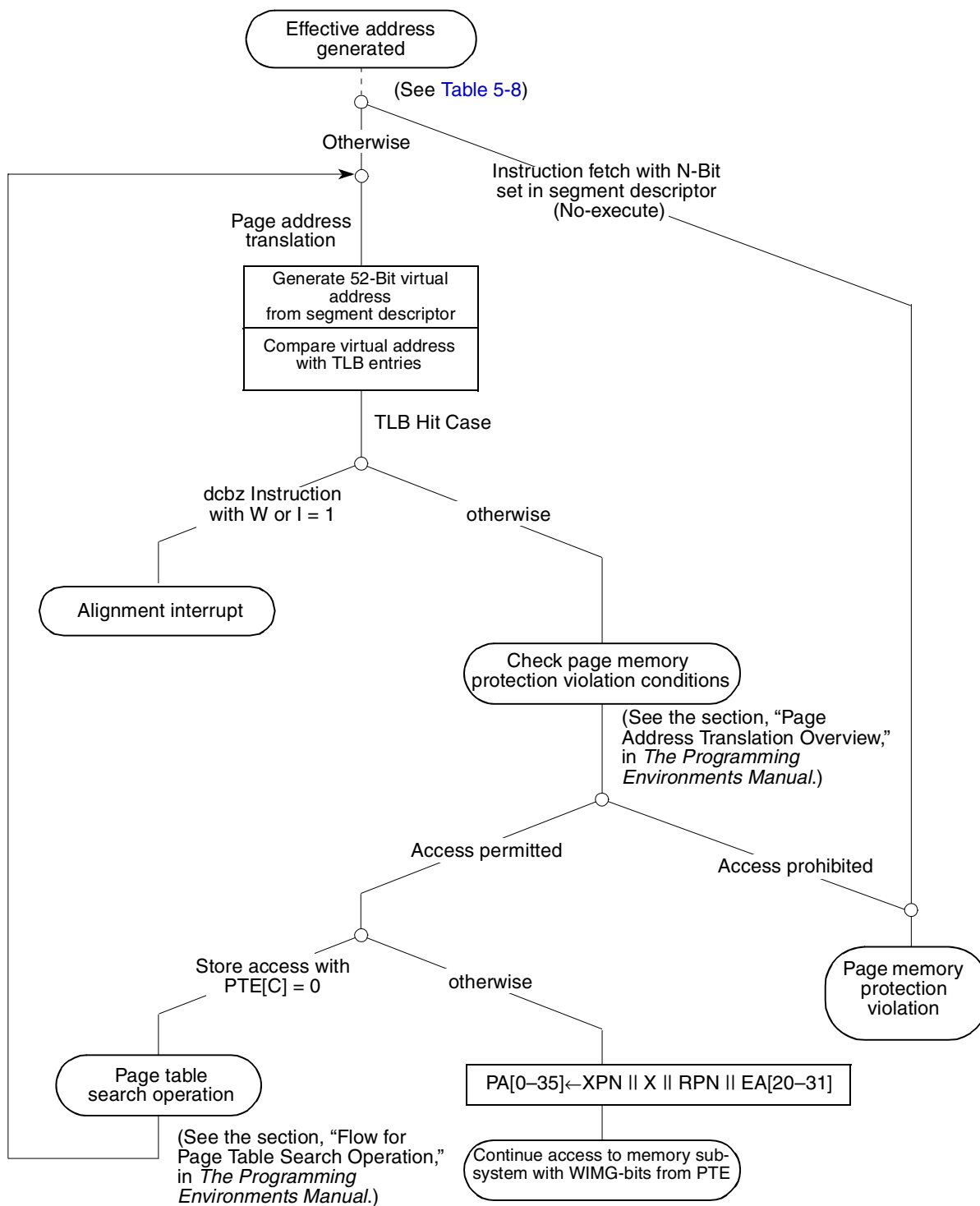


Figure 5-20. Page Address Translation Flow—TLB Hit—Extended Addressing

5.5 Hashed Page Tables—Extended Addressing

If a copy of the PTE corresponding to the VPN for an access is not resident in a TLB (corresponding to a miss in the TLB, provided a TLB is implemented), the processor must search (in hardware or software) for the PTE in the page tables set up by the operating system in main memory.

The algorithm specified by the architecture for accessing the page tables in hardware includes a hashing function on some of the virtual address bits. Thus the addresses for PTEs are allocated more evenly within the page tables and the hit rate of the page tables is maximized. This algorithm must be synthesized by the operating system for it to correctly place the page table entries in main memory.

When page table search operations are performed automatically by the hardware, they are performed using physical addresses and as if the memory access attribute bit $M = 1$ (memory coherency enforced in hardware). If the software performs the page table search operations, the accesses must be performed in real addressing mode ($MSR[DR] = 0$); this additionally guarantees that $M = 1$.

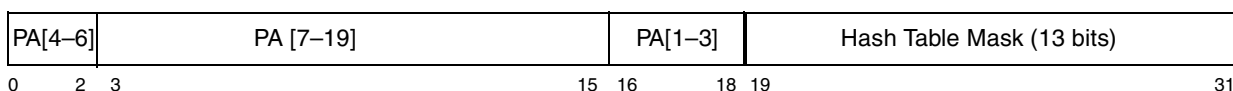
The section, “Hashed Page Tables,” in *The Programming Environments Manual* describes the format of the page tables and the algorithm used to access them for a 32-bit physical address. [Section 5.4.1.2, “Page Table Entry \(PTE\) Definition—Extended Addressing,”](#) describes the PTE format for extended addressing. The following subsections highlight the differences when translating for 36-bit physical addresses. In addition, the constraints imposed on the software in updating the page tables and the software table searching interrupt handlers (and other MMU resources) are described.

5.5.1 SDR1 Register Definition—Extended Addressing

The SDR1 register definition for 32-bit physical addressing is as described in Chapter 7, “Memory Management,” in *The Programming Environments Manual*. The SDR1 register contains the control information for the page table structure in that it defines the high-order bits for the physical base address of the page table and it defines the size of the table. Note that there are certain synchronization requirements for writing to SDR1 that are described in the section, “Synchronization Requirements for Special Registers and for Lookaside Buffers,” in *The Programming Environments Manual*. The format of the SDR1 register for extended addressing is described in the following sections. The SDR1 register has been modified for the e600 core to support extended 36-bit physical addresses (for when $HID0[XAEN = 1]$).

[Figure 5-21](#) shows the format of the SDR1 register in the bottom half of the figure; the top half shows how the physical address generated corresponds to SDR1 fields.

Physical Address Generated:



SDR1 Register:

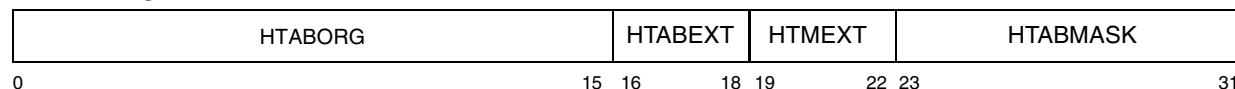


Figure 5-21. SDR1 Register Format—Extended Addressing

Bit settings for the SDR1 register are described in [Table 5-13](#).

Table 5-13. SDR1 Register Bit Settings—Extended Addressing

Bits	Name	Description
0–15	HTABORG	Physical base address of page table If $HID0[XAEN] = 1$, field contains physical address [4–19] If $HID0[XAEN] = 0$, field contains physical address [0–15]
16–18	HTABEXT	Extension bits for physical base address of page table If $HID0[XAEN] = 1$, field contains physical address [1–3] (and $PA0 = 0$) If $HID0[XAEN] = 0$, field is reserved
19–22	HTMEXT	Hash table mask extension bits If $HID0[XAEN] = 1$, field contains hash table mask [0–3] If $HID0[XAEN] = 0$, field is reserved
23–31	HTABMASK	Mask for page table address If $HID0[XAEN] = 1$, field contains hash table mask [4–12] If $HID0[XAEN] = 0$, field contains hash table mask [0–7]

SDR1 can be accessed with **mtspr** and **mfspir** using SPR 25. For synchronization requirements on the register see [Section 2.3.2.4, “Synchronization.”](#)

When extended addressing is disabled ($HID0[XAEN] = 0$), then the SDR1[HTABORG] field contains the high-order 16 bits of the 32-bit physical address of the page table. That is, SDR1[0–15] comprise the physical base address of the page table. Therefore, the beginning of the page table lies on a 2^{16} byte (64 Kbyte) boundary at a minimum. If extended addressing is enabled ($HID0[XAEN] = 1$), then a leading zero is concatenated with the values in the SDR1[HTABEXT] and SDR1[HTABORG] fields to produce the physical base address of the page table. In this case, the beginning of the page table lies on a 2^{19} (512 Kbyte) boundary at a minimum.

When extended addressing is enabled, a page table can be any size 2^n bytes where $16 \leq n \leq 29$. The HTMEXT field concatenated with the HTABMASK field in SDR1 contains a mask value that determines how many bits from the output of the hashing function are used as the page table index. This mask must be of the form 0b00...011...1 (a string of 0 bits followed by a string of 1 bits). As the table size increases, more bits are used from the output of the hashing function to index into the table. The 1 bits in HTMEXT || HTABMASK determine how many additional bits (beyond the minimum of 10) from the hash are used in the index; the HTABORG field must have the same number of low-order bits equal to 0 as the HTMEXT || HTABMASK fields have low-order bits equal to 1.

The SDR1[HTABEXT] field is ignored when extended addressing is disabled ($HID0[XAEN] = 0$). If extended addressing is enabled ($HID0[XAEN] = 1$), then the SDR1[HTABEXT] field contains bits 1-3 of the physical address of the page table. Note that bit 0 of the physical address of the page table is always 0.

5.5.1.1 Page Table Size

The number of entries in the page table directly affects performance because it influences the hit ratio in the page table and thus the rate of page fault interrupt conditions. If the table is too small, not all virtual pages that have physical page frames assigned may be mapped via the page table. This can happen if more than 16 entries map to the same primary/secondary pair of PTEGs; in this case, many hash collisions may occur.

In a 32-bit implementation, the minimum size for a page table is 64 Kbytes (2^{10} PTEGs of 64 bytes each). However, it is recommended that the total number of PTEGs in the page table be at least half the number of physical page frames to be mapped. While avoidance of hash collisions cannot be guaranteed for any size page table, making the page table larger than the recommended minimum size reduces the frequency of such collisions by making the primary PTEGs more sparsely populated, and further reducing the need to use the secondary PTEGs.

Table 5-14 shows some example sizes for total main memory with the e600 core using extended addressing. The recommended minimum page table size for these example memory sizes are then outlined, along with their corresponding HTABORG, HTMEXT, and HTABMASK settings in SDR1. Note that systems with less than 8 Mbytes of main memory may be designed with 32-bit processors, but the minimum amount of memory that can be used for the page tables in these cases is 64 Kbytes.

Table 5-14. Minimum Recommended Page Table Sizes—Extended Addressing

Total Main Memory	Recommended Minimum			Settings for Recommended Minimum	
	Memory for Page Tables	Number of Mapped Pages (PTEs)	Number of PTEGs	HTABORG (Maskable Bits 3–15)	HTMEXT HTABMASK SDR1[19-31]
8 Mbytes (2^{23})	64 Kbytes (2^{16})	2^{13}	2^{10}	x xxxx xxxx xxxx	0 0000 0000 0000
16 Mbytes (2^{24})	128 Kbytes (2^{17})	2^{14}	2^{11}	x xxxx xxxx xxx0	0 0000 0000 0001
32 Mbytes (2^{25})	256 Kbytes (2^{18})	2^{15}	2^{12}	x xxxx xxxx xx00	0 0000 0000 0011
64 Mbytes (2^{26})	512 Kbytes (2^{19})	2^{16}	2^{13}	x xxxx xxxx x000	0 0000 0000 0111
128 Mbytes (2^{27})	1 Mbyte (2^{20})	2^{17}	2^{14}	x xxxx xxxx 0000	0 0000 0000 1111
256 Mbytes (2^{28})	2 Mbytes (2^{21})	2^{18}	2^{15}	x xxxx xxx0 0000	0 0000 0001 1111
512 Mbytes (2^{29})	4 Mbytes (2^{22})	2^{19}	2^{16}	x xxxx xx00 0000	0 0000 0011 1111
1 Gbytes (2^{30})	8 Mbytes (2^{23})	2^{20}	2^{17}	x xxxx x000 0000	0 0000 0111 1111
2 Gbytes (2^{31})	16 Mbytes (2^{24})	2^{21}	2^{18}	x xxxx 0000 0000	0 0000 1111 1111
4 Gbytes (2^{32})	32 Mbytes (2^{25})	2^{22}	2^{19}	x xxx0 0000 0000	0 0001 1111 1111
8 Gbytes (2^{33})	64 Mbytes (2^{26})	2^{23}	2^{20}	x xx00 0000 0000	0 0011 1111 1111
16 Gbytes (2^{34})	128 Mbytes (2^{27})	2^{24}	2^{21}	x x000 0000 0000	0 0111 1111 1111
32 Gbytes (2^{35})	256 Mbytes (2^{28})	2^{25}	2^{22}	x 0000 0000 0000	0 1111 1111 1111
64 Gbytes (2^{36})	512 Mbytes (2^{29})	2^{26}	2^{23}	0 0000 0000 0000	1 1111 1111 1111

As an example, if the physical memory size is 2^{35} bytes (32 Gbyte), then there are $2^{35} - 2^{12}$ (4 Kbyte page size) = 2^{23} (8 Mbyte) total page frames. If this number of page frames is divided by 2, the resultant minimum recommended page table size is 2^{22} PTEGs, or 2^{28} bytes (256 Mbytes) of memory for the page tables.

5.5.1.2 Page Table Hashing Functions

The MMU uses two different hashing functions, a primary and a secondary, in the creation of the physical addresses used in a page table search operation. These hashing functions distribute the PTEs within the page table, in that there are two possible PTEGs where a given PTE can reside. Additionally, there are eight possible PTE locations within a PTEG where a given PTE can reside. If a PTE is not found using the primary hashing function, the secondary hashing function is performed, and the secondary PTEG is searched. Note that these two functions must also be used by the operating system to set up the page tables in memory appropriately.

The address of a PTEG is derived from the HTABORG field of the SDR1 register, and the output of the corresponding hashing function (primary hashing function for primary PTEG and secondary hashing function for a secondary PTEG). The values in the HTMEXT and HTABMASK fields determine how many of the high-order hash value bits are masked and how many are used in the generation of the physical address of the PTEG.

Figure 5-22 depicts the hashing functions used by the e600 core to generate a 36-bit physical table entry group address. The inputs to the primary hashing function are the low-order 23 bits of the VSID field of the selected segment register (VA[1–23]), and the page index field of the effective address (VA[24–39]) concatenated with seven zero high-order bits. The XOR of these two values generates the output of the primary hashing function (hash value 1).

When the secondary hashing function is required, the output of the primary hashing function is complemented with one's complement arithmetic, to provide hash value 2.

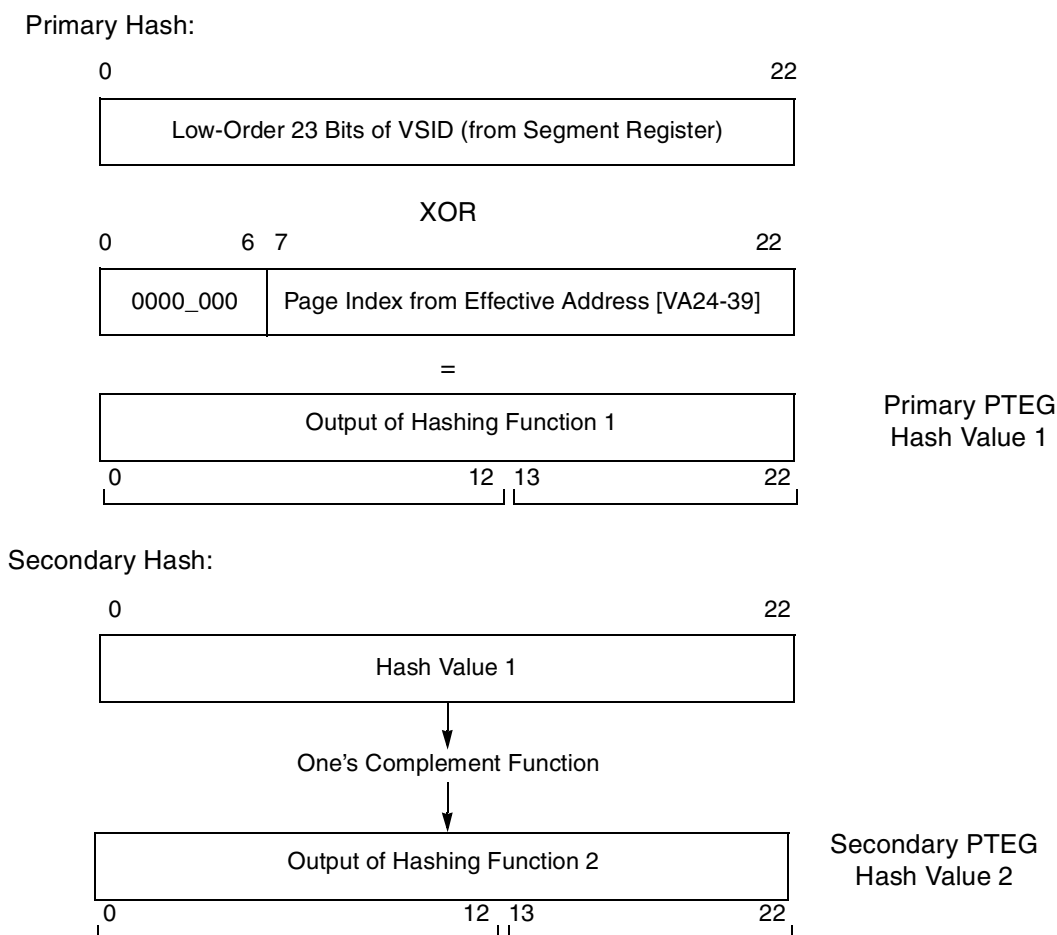


Figure 5-22. Hashing Functions for Page Table Entry Group Address

5.5.1.3 Page Table Address Generation

The following sections illustrate the generation of the addresses used for accessing the hashed page tables. As stated earlier, the operating system must synthesize the table search algorithm for setting up the tables. This process is as described in Chapter 7, “Memory Management,” in *The Programming Environments Manual*.

For extended addressing, PTEG[0] is zero and PTEG[1–3] is defined by the HTABEXT field of SDR1 (SDR1[16–18]) as shown in Figure 5-24. PTEG[4–6] is defined by the highest order bits of the HTABORG field (SDR1[0–2]). PTEG[7–19] are derived from the masking of the high-order bits of the hash value[0–12] with SDR1[HTABMASK] and SDR1[HTMEXT]. The value from the AND function is then concatenated with (implemented as an OR function) the high-order bits of the unmasked HTABORG bits SDR1[3–15]. PTEG[20–29] are the 10 low-order bits of the hash value. PTEG[30–35] are zeros. In the process of searching for a PTE, the processor checks up to eight PTEs located in the primary PTEG and up to eight PTEs located in the secondary PTEG, if required, searching for a match.

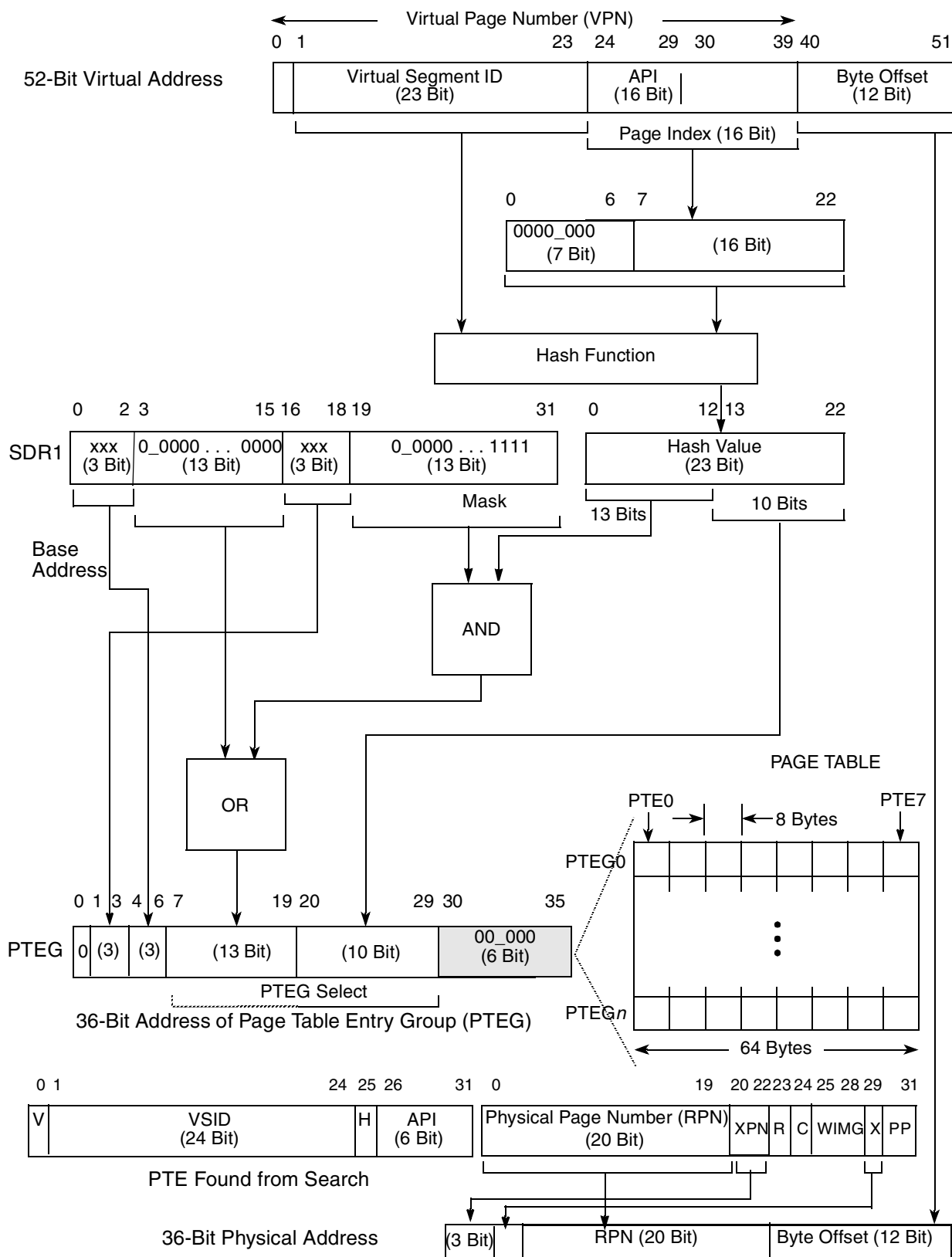


Figure 5-23. PTEG Address Generation for a Page Table Search—Extended Addressing

these two bits are generated from bits 11–12 of the output of the hashing function, logically ORed with bits 14–15 of the HTABORG field (that must be zero). If bits 18–29 of PTEGaddr1 were derived by using the primary hashing function, then PTEGaddr2 corresponds to the secondary PTEG.

Note, however, that bits 18–29 in PTEGaddr2 can also be derived from a combination of effective address bits, segment register bits, and the primary hashing function. In this case, PTEGaddr1 corresponds to the secondary PTEG. Thus while a PTEG may be considered a primary PTEG for some effective addresses (and segment register bits), it may also correspond to the secondary PTEG for a different effective address (and segment register value).

It is the value of the H bit in each of the individual PTEs that identifies a particular PTE as either primary or secondary (there may be PTEs that correspond to a primary PTEG and PTEs that correspond to a secondary PTEG, all within the same physical PTEG address space). Thus only the PTEs that have H = 0 are checked for a hit during a primary PTEG search. Likewise, only PTEs with H = 1 are checked in the case of a secondary PTEG search.

5.5.1.5 PTEG Address Mapping Examples—Extended Addressing

This section contains two examples of an effective address and how its address translation (the PTE) maps into the primary PTEG in physical memory. The examples illustrate how the processor generates PTEG addresses for a table search operation; this is also the algorithm that must be used by the operating system in creating page tables.

Figure 5-26 shows an example of PTEG address generation for extended addressing. In the example, the value in SDR1 defines a page table at address 0x4_0F98_0000 that contains 8192 PTEGs. The example effective address selects segment register 0 (SR0) with the highest-order four bits. The contents of SR0 are then used along with bits 4–31 of the effective address to create the 52-bit virtual address.

To generate the address of the primary PTEG, bits 1–23, and bits 24–39 of the virtual address are then used as inputs into the primary hashing function (XOR) to generate hash value 1. The low-order 13 bits of hash value 1 are then concatenated with the high-order 13 bits of HTABORG and HTBEXT with an added leading zero. Finally the address is appended with six low-order 0 bits, defining the address of the primary PTEG (0x4_0F9F_F980).

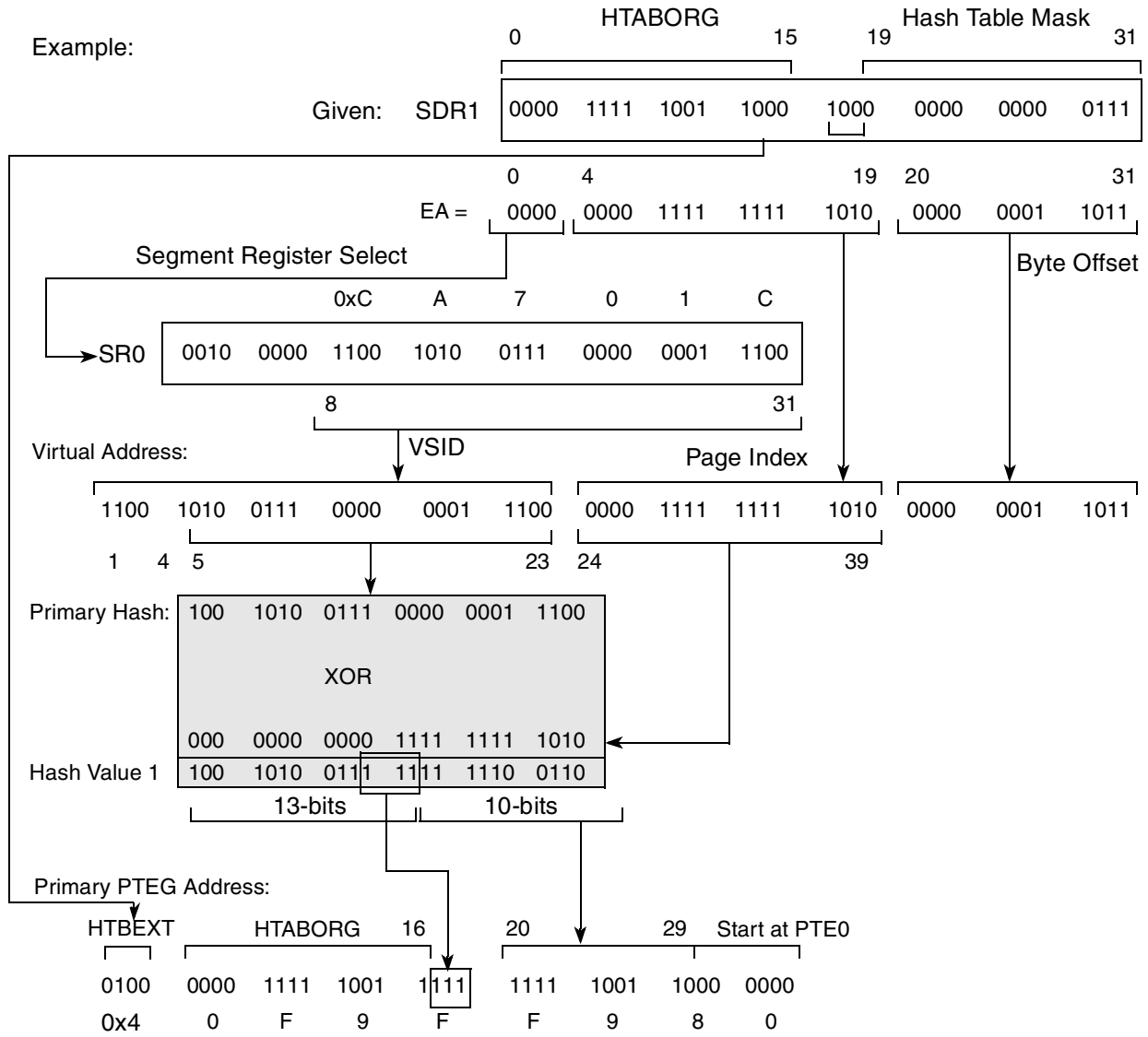


Figure 5-25. Example Primary PTEG Address Generation

Figure 5-26 shows the generation of the secondary PTEG address for this example. If the secondary PTEG is required, the secondary hash function is performed and the low-order 13 bits of hash value 2 are then ORed with the high-order 16 bits of HTABORG (bits 13–15 should be zero), and HTBEXT with an added leading zero. Finally, the address is appended with six low-order 0 bits, defining the address of the secondary PTEG (0x4_0F98_0640).

As described in Figure 5-23, the 10 low-order bits of the page index field are always used in the generation of a PTEG address (through the hashing function) for a 32-bit implementation. This is why only the abbreviated page index (API) is defined for a PTE (the entire page index field does not need to be checked). For a given effective address, the low-order 10 bits of the page index (at least) contribute to the PTEG address (both primary and secondary) where the corresponding PTE may reside in memory. Therefore, if

the high-order 6 bits (the API field) of the page index match with the API field of a PTE within the specified PTEG, the PTE mapping is guaranteed to be the unique PTE required.

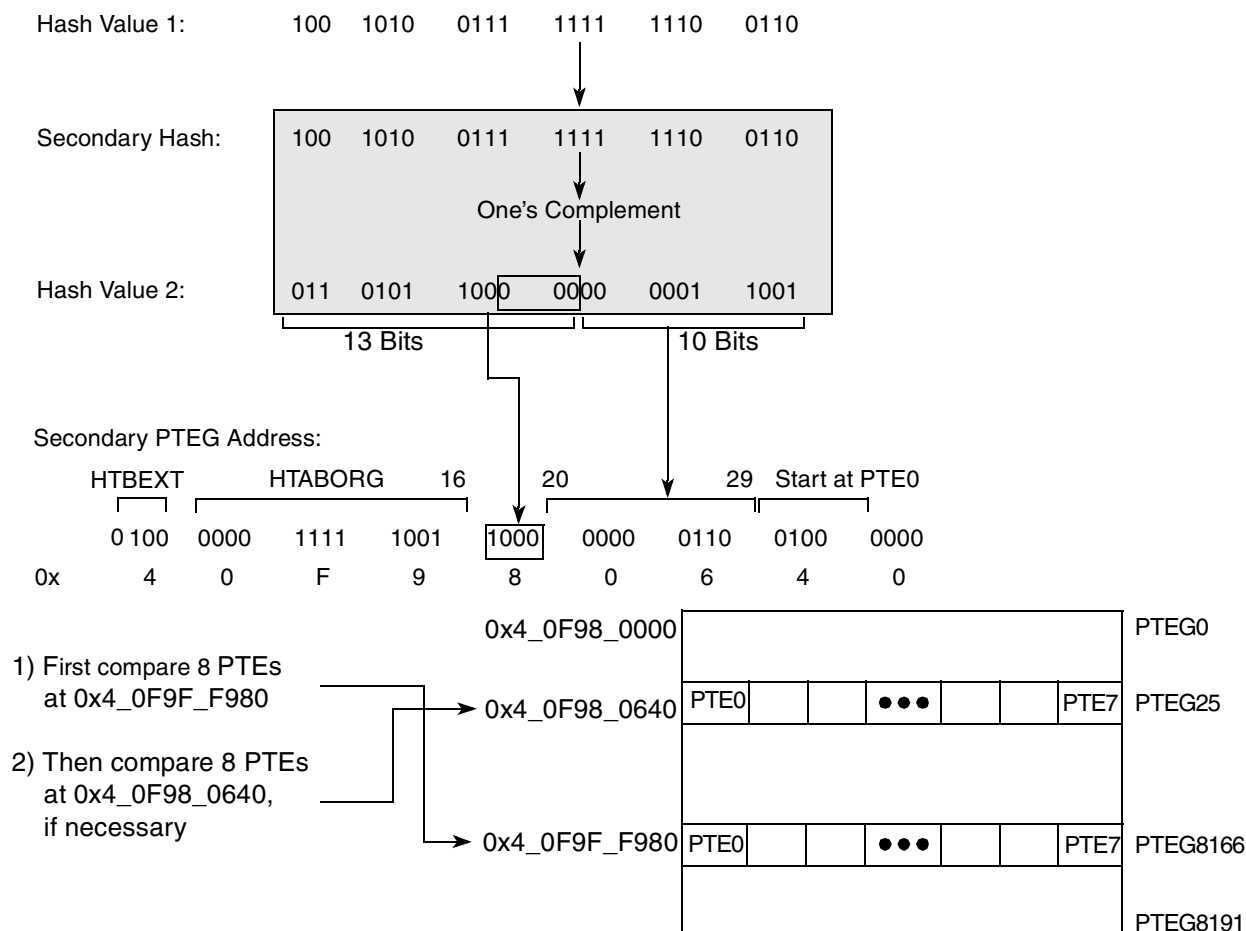


Figure 5-26. Example Secondary PTEG Address Generation

Note that a given PTEG address does not map back to a unique effective address. Not only can a given PTEG be considered both a primary and a secondary PTEG (as described in Section 5.5.1.4, “Page Table Structure Example—Extended Addressing”), but in this example, bits 24–26 of the page index field of the virtual address are not used to generate the PTEG address. Therefore, any of the eight combinations of these bits will map to the same primary PTEG address. (However, these bits are part of the API and are therefore compared for each PTE within the PTEG to determine if there is a hit.) Furthermore, an effective address can select a different segment register with a different value such that the output of the primary (or secondary) hashing function happens to equal the hash values shown in the example. Thus these effective addresses would also map to the same PTEG addresses shown.

5.5.2 Page Table Search Operations—Implementation

If the translation is not found in the TLBs (a TLB miss), the e600 core initiates a hardware or software table search operation as described in this section for 36-bit addressing. Formats for the PTEs used in 32-bit addressing are described in “PTE Format for 32-Bit Implementations,” in Chapter 7, “Memory Management,” of *The Programming Environments Manual*.

5.5.2.1 Conditions for a Page Table Search Operation

For instruction accesses, the processor containing an e600 core does not initiate a table search operation for an ITLB miss until the completion buffer is empty and the completed store queue is empty. Also, the instruction buffer must be empty, there must be no other interrupts pending, there must be no branch processing in progress, and there must be no outstanding instruction cache misses.

Also, the MMU does not perform a hardware table search due to DTLB misses (or to modify the C bit) until the access is absolutely required by the program flow and there are no other interrupts pending.

In the e600 core, a TLB miss (and subsequent page table search operation) occurs transparently to the program. Thus if a TLB miss occurs when a misaligned access crosses a translation boundary, the second portion of the misaligned access is completed automatically once the table search operation completes successfully. If the table search operation results in a page fault, an interrupt occurs and upon returning from the page fault handling routine, the entire misaligned access is restarted beginning with the first portion of the access.

Note that, as described in [Chapter 6, “Instruction Timing,”](#) store gathering does not occur while a page table search operation is in progress.

The AltiVec data stream touch instructions (**dst[t]** and **dstst[t]**) provide the ability to prefetch up to 128 Kbytes of data per instruction. As described in [Chapter 6, “Instruction Timing,”](#) a **dst[t]** or **dstst[t]** instruction can be retired from the completion buffer as soon as the instruction is loaded into the vector touch queue (VTQ). However, if a line fetch in the VTQ requires a table search operation before the instruction is retired, then the table search operation is delayed until the instruction is retired. If a line fetch in the VTQ requires a table search operation after the instruction has been retired, the table search operation is initiated immediately.

To further increase performance, the VTQ stream engines operate in parallel with the other execution units. Also, the TLBs are non-blocking, and are available to the instruction unit and LSU for both instruction and data address translation during a VTQ-initiated table search operation.

5.5.2.2 AltiVec Line Fetch Skipping

As described in [Chapter 7, “AltiVec Technology Implementation,”](#) there are many conditions (interrupts, etc.) that cause the stream fetch performed by a VTQ stream engine to abort. In the case of a VTQ-initiated table search operation, when an interrupt or exception condition occurs, the stream engine pauses, the line-fetch that caused the table search operation is effectively dropped, and no MMU interrupts are reported for this line-fetch. When the stream engine resumes operation, the next line fetch is attempted, causing a skip of one line fetch in the stream engine.

Also, when a **tlbsync** instruction is executed while a VTQ-initiated table search operation is in progress, that table search operation is aborted, potentially causing a line fetch skip.

5.5.2.3 Page Table Search Operation—Conceptual Flow

The following is a summary of the page table search process performed automatically by the e600 core when hardware table searching is enabled. A very similar flow occurs when the software table searching is enabled.

1. The 32-bit physical address of the primary PTEG is generated as described in Chapter 7, “Memory Management,” of *The Programming Environments Manual*. When extended addressing is enabled, the 36-bit address generation is described in [Section 5.5.1.3, “Page Table Address Generation.”](#)
2. The first PTE (PTE0) in the primary PTEG is read from memory. PTE reads occur with an implied WIMG memory/cache mode control bit setting of 0b001x. Therefore, they are considered cacheable and read (burst) from memory and placed in the cache. Because the table search operation is never speculative and is cacheable, the G-bit has no effect.
3. The PTE in the selected PTEG is tested for a match with the virtual page number (VPN) of the access. The VPN is the VSID concatenated with the page index field of the virtual address. For a match to occur, the following must be true:
 - PTE[H] = 0
 - PTE[V] = 1
 - PTE[VSID] = VA[0–23]
 - PTE[API] = VA[24–29]
4. If a match is not found, step 3 is repeated for each of the other seven PTEs in the primary PTEG. If a match is found, the table search process continues as described in step 8. If a match is not found within the 8 PTEs of the primary PTEG, the address of the secondary PTEG is generated.
5. The first PTE (PTE0) in the secondary PTEG is read from memory. Again, because PTE reads have a WIMG bit combination of 0b001x, an entire cache line is read into the cache. The PTE in the selected secondary PTEG is tested for a match with the virtual page number (VPN) of the access. For a match to occur, the following must be true:
 - PTE[H] = 1
 - PTE[V] = 1
 - PTE[VSID] = VA[0–23]
 - PTE[API] = VA[24–29]
6. If a match is not found, step 6 is repeated for each of the other seven PTEs in the secondary PTEG. If it is never found, an interrupt is taken (step 9).
7. If a match is found, the PTE is written into the TLB in the core and the R bit is updated in the PTE in memory (if necessary). If there is no memory protection violation, the C bit is also updated in memory (if the access is a write operation) and the table search is complete.

8. If a match is not found within the 8 PTEs of the secondary PTEG, the search fails, and a page fault exception condition occurs (either an ISI or a DSI interrupt). Note that the software routines that implement this algorithm for the e600 core must synthesize this condition by appropriately setting the bits in SRR1 (or DSISR) and branching to the ISI or DSI handler routine.

Reads from memory for hardware table search operations are performed as global (but not exclusive), cacheable operations, and can be loaded into the cache inside the core. These types of transactions should be generated when software table searching is enabled.

Figure 5-27 and Figure 5-28 show how the conceptual flow diagrams for the primary and secondary page table search operations, described in the section, “Page Table Search Operation,” in *The Programming Environments Manual*, are realized in the e600 core. Recall that the architecture allows for implementations to perform the page table search operations automatically (in hardware), or software assistance may be allowed, as is an option with the e600 core.

Figure 5-27 shows the case of a **dcbz** instruction that is executed with $W = 1$ or $I = 1$, and that the R bit may be updated in memory (if required) before the operation is performed or the alignment interrupt occurs. The R bit may also be updated if memory protection is violated.

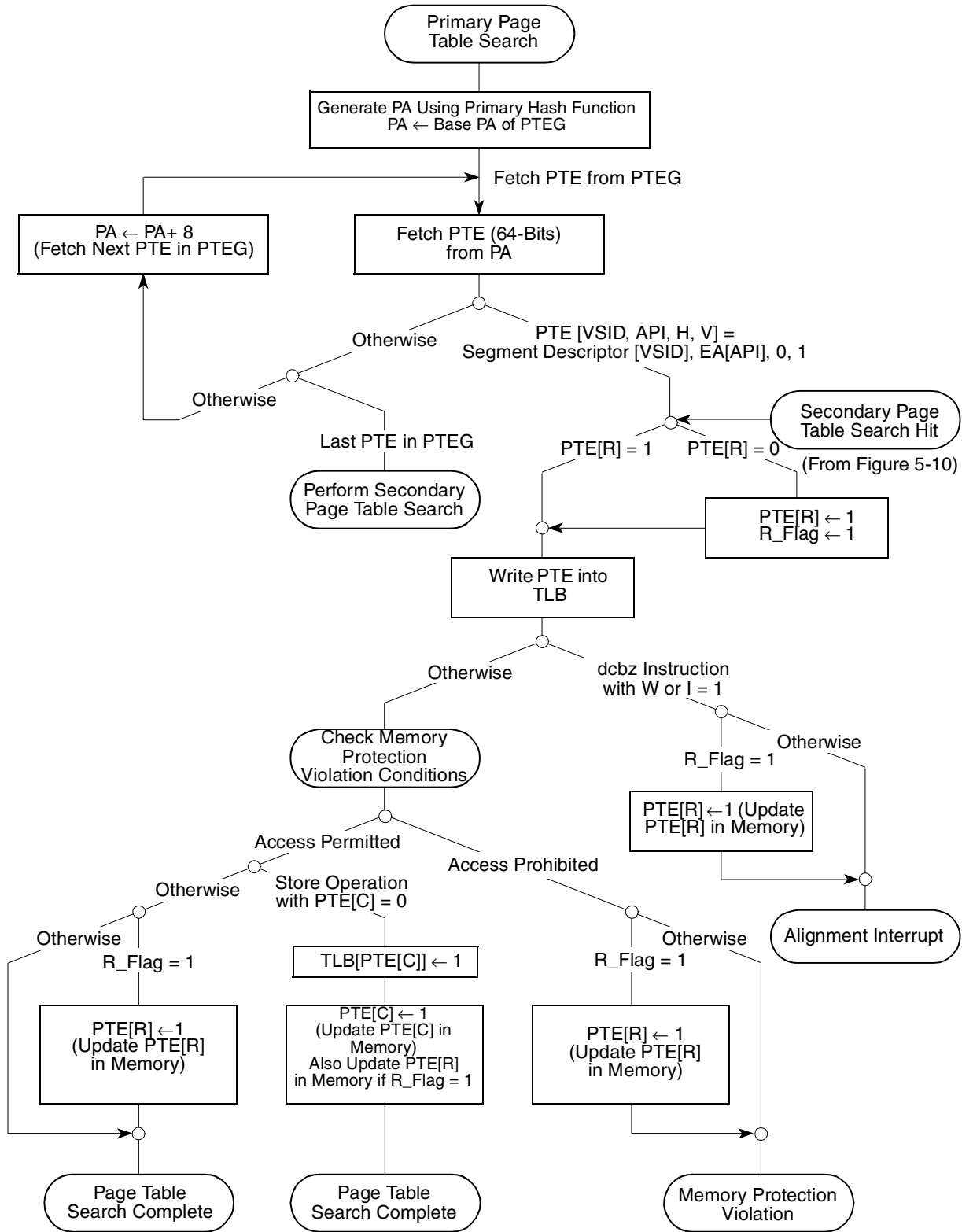


Figure 5-27. Primary Page Table Search—Conceptual Flow

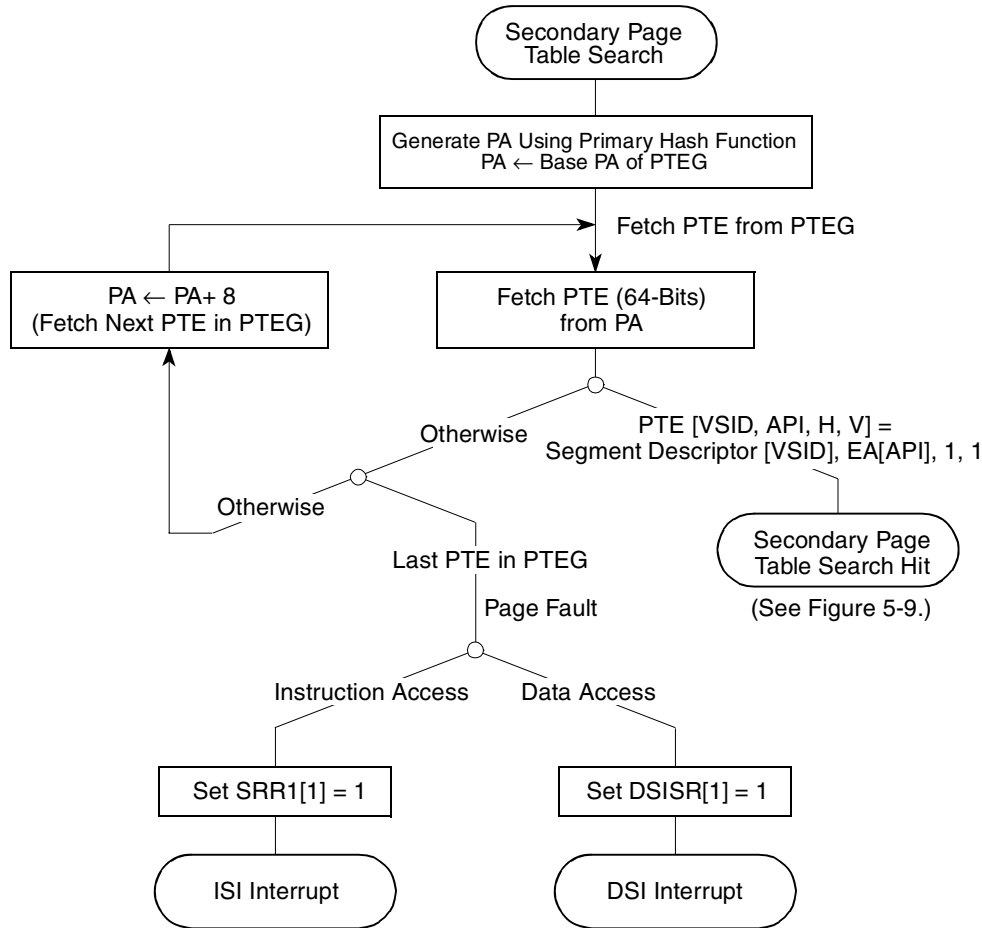


Figure 5-28. Secondary Page Table Search Flow—Conceptual Flow

5.5.3 Page Table Updates

When TLBs are implemented (as in the e600 core) they are defined as noncoherent caches of the page tables. TLB entries must be flushed explicitly with the TLB invalidate entry instruction (**tlbie**) whenever the corresponding PTE is modified.

Chapter 7, “Memory Management,” in *The Programming Environments Manual* describes some required sequences of instructions for modifying the page tables. In a multiprocessor e600 core environment, PTEs can only be modified by adhering to the procedure for deleting a PTE, followed by the procedure for adding a PTE.

Thus the following code should be used:

```

/* Code for Modifying a Page Table Entry */
/* First delete the current page table entry */
PTEV ← 0/* (other fields don't matter) */
sync          /* ensure update completed */
tlbie(old_EA) /* invalidate old translation */
eieio        /* order tlbie before tlbsync */
tlbsync      /* ensure tlbie completed on all processors */
sync         /* ensure tlbsync completed */
  
```

```

/* Then add new PTE over old */
PTERPN,R,C,WIMG,PP <- new values
eieio /* order 1st PTE update before 2nd */
PTEVSID,API,H,V <- new values (V=1)
sync /* ensure updates completed */

```

Processors may write referenced and changed bits with unsynchronized, atomic byte store operations. Note that the V, R, and C bits each reside in a distinct byte of a PTE. Therefore, extreme care must be taken to use byte writes when updating only one of these bits.

Explicitly altering certain MSR bits (using the `mtmsr` instruction), or explicitly altering PTEs, or certain system registers, may have the side effect of changing the effective or physical addresses from which the current instruction stream is being fetched. This kind of side effect is defined as an implicit branch. Implicit branches are not supported and an attempt to perform one causes boundedly-undefined results. Therefore, PTEs must not be changed in a manner that causes an implicit branch. Chapter 2, “PowerPC Register Set,” in *The Programming Environments Manual*, lists the possible implicit branch conditions that can occur when system registers and MSR bits are changed.

5.5.4 Segment Register Updates

Synchronization requirements for using the move to segment register instructions are described in “Synchronization Requirements for Special Registers and for Lookaside Buffers” in Chapter 2, “PowerPC Register Set,” in *The Programming Environments Manual*.

5.5.5 Implementation-Specific Software Table Search Operation

The e600 core has a set of implementation-specific registers, interrupts, and instructions that facilitate very efficient software searching of the page tables in memory for when software table searching is enabled (`HID0[STEN] = 1`). This section describes those resources and provides three example code sequences that can be used in a e600 core system for an efficient search of the translation tables in software. These code sequences can be used as handlers for the three interrupts requiring access to the PTEs in the page tables in memory in this case—instruction TLB miss, data TLB miss on load, and data TLB miss on store interrupts.

5.5.5.1 Resources for Table Search Operations

When software table searching is enabled, the system software must set up the translation page tables in memory, and assist the processor in loading PTEs into the TLBs within the core. When a required TLB entry is not found in the appropriate TLB, the processor vectors to one of the three TLB miss interrupt handlers so that the software can perform a table search operation and load the TLB. When this occurs, the processor automatically saves information about the access and the executing context. [Table 5-15](#) provides a summary of the implementation-specific interrupts, registers, and instructions, that can be used by the TLB miss interrupt handler software in e600 core systems. Refer to [Chapter 4, “Interrupts,”](#) for more information about interrupt processing.

Table 5-15. Implementation-Specific Resources for Software Table Search Operations

Resource	Name	Description
Interrupts	ITLB miss interrupt (offset 0x1000)	No matching entry found in ITLB
	DTLB miss on load interrupt (offset 0x1100)	No matching entry found in DTLB for a load data access
	DTLB miss on store interrupt—also caused when changed bit must be updated (offset 0x1200)	No matching entry found in DTLB for a store data access or matching DLTB entry has C = 0 and access is a store.
Registers	TLBMISS	When either an instruction TLB miss, data TLB miss on load, and data TLB miss on store interrupt occurs, the TLBMISS register contains part of the effective address of the instruction or data access that caused the miss interrupt.
	PTEHI	When software table searching is enabled (HID0[STEN] = 1), and a TLB miss interrupt occurs, the fields of the PTEHI register are loaded automatically with the VSID information from the corresponding SR, and the API of the miss address. The PTEHI register is also used by the tibli and tibld instructions.
	PTELO	When software table searching is enabled (HID0[STEN] = 1), and a TLB miss interrupt occurs, software determines the lower 32 bits of the PTE and places those bits in the PTELO register. The PTELO register is also used by the tibli and tibld instructions.
	SPRG4–7 ¹	When software table searching is enabled (HID0[STEN] = 1), and a TLB miss interrupt occurs, the SPRGs provide additional registers to be used by system software for table software searching.
Instructions	tibli rB	Loads the contents of the PTEHI and PTELO registers into the ITLB entry selected by <EA> where <EA> = bits 10–19 of rB. Way to be loaded is selected by rB[31] (LRU way bit).
	tibld rB	Loads the contents of the PTEHI and PTELO registers into the DTLB entry selected by <EA> where <EA> = bits 10–19 of rB. Way to be loaded is selected by rB[31] (LRU way bit).

¹ Specific to the e600 core.

In addition, the e600 core contains the following features that do not specifically control the e600 core MMU but that are implemented to increase performance and flexibility in the software table search routines whenever one of the three TLB miss interrupts occurs:

- TLBMISS[31] identifies the associativity class of the TLB entry selected for replacement by the LRU algorithm. The software can change this value, effectively overriding the replacement algorithm. In the case of a store hit with C = 0, TLBMISS[31] points to the way that missed on the store access (and not the entry that hit with C = 0). Therefore, software must toggle this bit before placing it into rB[31]. Then **tibld** rB is executed by software, updating the entry that originally hit with C = 0.

- The SRR1[KEY] bit is used by the table search software to determine if there is a protection violation associated with the access (useful on data write misses for determining if the C bit should be updated in the table). [Table 5-16](#) summarizes the SRR1 bits updated whenever one of the three TLB miss interrupts occurs.

Table 5-16. Implementation-Specific SRR1 Bits

Bit Number	Name	Function
11	CEQ0	Set if the interrupt was caused by the a store to a page with PTE[C] = 0.
12	KEY	Key for TLB miss (either SR[Ks] or SR[Kp] from the segment register, depending on whether the access is a supervisor or user access)

The key bit saved in SRR1 is derived as shown in [Figure 5-29](#).

Select KEY from segment register:
 If MSR[PR] = 0, KEY = Ks
 If MSR[PR] = 1, KEY = Kp

Figure 5-29. Derivation of Key Bit for SRR1

The remainder of this section describes the format of the implementation-specific SPRs that are not defined by the PowerPC architecture, but that are used by the TLB miss interrupt handlers. These registers can be accessed by supervisor-level instructions only. Any attempt to access these SPRs with user-level instructions results in a privileged instruction program interrupt. As TLBMISS, PTEHI, and PTELO are used to access the translation tables for software table search operations, they should only be accessed when address translation is disabled (that is, MSR[IR] = 0 and MSR[DR] = 0). Note that MSR[IR] and MSR[DR] are cleared by the processor whenever an interrupt occurs.

Software must ensure that a TLB lookup never results in a match on both ways of the same set. It is a programming error for multiple ways to match and it can produce unpredictable results. Software is required to keep track of the current contents of the TLBs.

In a multiprocessing system, software must take steps to ensure coherency during a software table search operation. If a processor executes a **tlbie** instruction while another processor is handling a software table search interrupt, coherency can be lost and the TLB could be corrupted. A semaphore mechanism should be used when performing a software table search operation in a multiprocessing environment to ensure that coherency is maintained.

5.5.5.1.1 TLB Miss Register (TLBMISS)

The TLBMISS register is automatically loaded by the e600 core when software searching is enabled (HID0[XAEN] = 1) and a TLB miss interrupt occurs. Its contents are used by the TLB miss interrupt handlers (the software table search routines) to start the search process. Note that the e600 core always loads a big-endian address into the TLBMISS register. This register is read-only. The TLBMISS register has the format shown in [Figure 5-30](#).

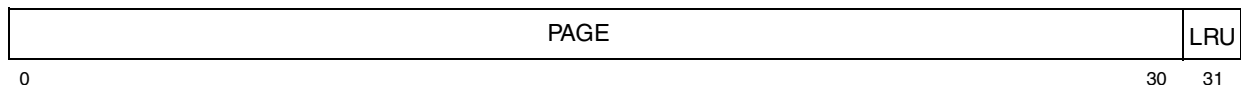


Figure 5-30. TLBMISS Register

Table 5-17 described the bits in the TLBMISS register.

Table 5-17. TLBMISS Register—Field and Bit Descriptions

Bits	Name	Function
0–30	PAGE	Effective page address. Stores EA[0–30] of the access that caused the TLB miss interrupt.
31	LRU	Least recently used way of the addressed TLB set. The LRU bit can be loaded into bit 31 of rB, before execution of tbli or tblid to select the way to be replaced for a TLB miss. However, this value should be inverted in rB prior to execution of tbli or tblid for a TLB miss interrupt caused by the need to update the C-bit.

5.5.5.1.2 Page Table Entry Registers (PTEHI and PTELO)

The PTEHI and PTELO registers are used by the **tblid** and **tbli** instructions to create a TLB entry when extended addressing is enabled (HID0[XAEN] = 1). When software table searching is enabled (HID0[STEN] = 1), and a TLB miss interrupt occurs, the bits of the page table entry (PTE) for this access are located by software and saved in the PTE registers. Figure 5-31 shows the format for two supervisor registers PTEHI and PTELO, respectively.

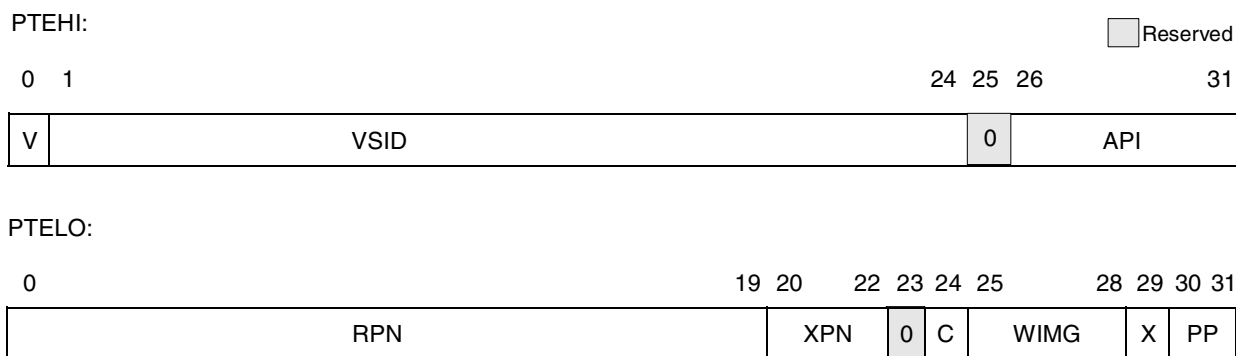


Figure 5-31. PTEHI and PTELO Registers—Extended Addressing

Note that the contents of PTEHI are automatically loaded when any of the three software table search interrupts is taken. PTELO is loaded by the software table search routines (the TLB miss interrupt handlers) based on the valid PTE located in the page tables prior to execution of **tbli** or **tblid** instruction.

Table 5-18 lists the corresponding bit definitions for the PTEHI and PTELO registers.

Table 5-18. PTEHI and PTELO Bit Definitions

Word	Bit	Name	Description
PTEHI	0	V	Entry valid (V = 1) or invalid (V = 0). Always set by the processor on a TLB miss interrupt.
	1–24	VSID	Virtual segment ID. The corresponding SR[VSID] field is copied to this field.
	25	—	Reserved
	26–31	API	Abbreviated page index. TLB miss interrupts will set this field with bits from TLBMISS[4–9] which are bits from the effective address for the access that caused the software table search operation. The tlbli and tlbld instructions will ignore the API bits in PTEHI register and get the API from instruction's operand, rB . However, for future compatibility, the API in rB should match the PTEHI[API].
PTELO	0–19	RPN	Physical page number
	20–22	XPN	Extended page number. The XPN field provides the physical address bits, PA[0–2].
	23	—	Reserved
	24	C	Changed bit
	25–28	WIMG	Memory/cache control bits
	29	X	Extended page number. The X field provides the physical address bit 3, PA[3].
	30–31	PP	Page protection bits

Note that PTELO[23] corresponds to the referenced bit in a PTE. The referenced bit is not stored in the page tables, so this bit is ignored in the PTELO register. All the other bits in PTELO correspond to the bits in the low word of the PTE. When extended addressing is not enabled, (HID0[XAEN] = 0), the PTELO[XPI] and PTELO[X] values should be zeros so that the four most-significant bits of the physical address are zeros.

5.5.5.1.3 Special Purpose Registers (4–7)

Four additional SPRGs are provided on the e600 core. The registers are provided to assist in a software table search. For example, in the example code in [Section 5.5.5.2.2, “Code for Example Interrupt Handlers,”](#) the register values are saved into the SPRGs to avoid any latency in storing the values out to memory. Thus using the additional SPRGs made the code faster and simpler.

5.5.5.2 Example Software Table Search Operation

When a TLB miss occurs, the instruction or data MMU loads the TLBMISS register with the effective address (EA[0–30]) of the access. The processor completes all instructions dispatched before the interrupt, status information is saved in SRR1, and one of the three TLB miss interrupts is taken.

The software uses whatever routine it implemented to generate the PTE. Then it places the upper and lower portions of the PTE into PTEHI and PTELO, respectively. Then it uses the **tlbli** or **tlbld** instructions to load the contents of the PTE into the selected TLB entry. The TLB entry is selected by bits 10–19 of **rB** and the way is selected by bit 31 of **rB**.

Note that a miss caused by a **dcbt**, **dst**, or **dstst** instruction while $HID0[STEN] = 1$ does not cause one of the software table searching interrupts; the **dcbt** in this case functions as a no-op and **dst/dstst** cause the stream to terminate.

If the PTE search algorithm does not produce a desired PTE, a page fault interrupt must be synthesized. Thus the appropriate bits must be set in SRR1 (or DSISR) and the TLB miss handler must branch to either the ISI or DSI interrupt handler, that handles the page fault condition.

This section provides a flow diagram outlining an example software algorithm that mimics the hardware table search procedure used by the e600 core (and other processors and cores that implement the PowerPC architecture). This software can be used to handle the three TLB miss interrupts. Some example assembly language that implements that flow is also provided. However, software can implement other types of page tables and PTE search algorithms using the same resources.

5.5.5.2.1 Flow for Example Interrupt Handlers

[Figure 5-32](#) shows the flow for the example TLB miss interrupt handlers. [Figure 5-33](#) shows the flow for how a PTEG address is generated. The flow shown is common for the three interrupt handlers. Also, in the cases of store instructions that cause either a TLB miss or require a table search operation to update the C bit, the flow shows that the C bit is set in both the TLB entry and the PTE in memory. Note that in the case of a page fault (no PTE found in the table search operation), the setup for the ISI or DSI interrupt is slightly different.

[Figure 5-34](#) shows the flow for checking the R and C bits and setting them appropriately, [Figure 5-35](#) shows the flow for synthesizing a page fault interrupt when no PTE is found. [Figure 5-36](#) shows the flow for managing the cases of a TLB miss on an instruction access to guarded memory, and a TLB miss when $C = 0$ and a protection violation exists. The setup for these protection violation interrupts is very similar to that of page fault conditions (as shown in [Figure 5-35](#)) except that different bits in SRR1 (and DSISR) are set.

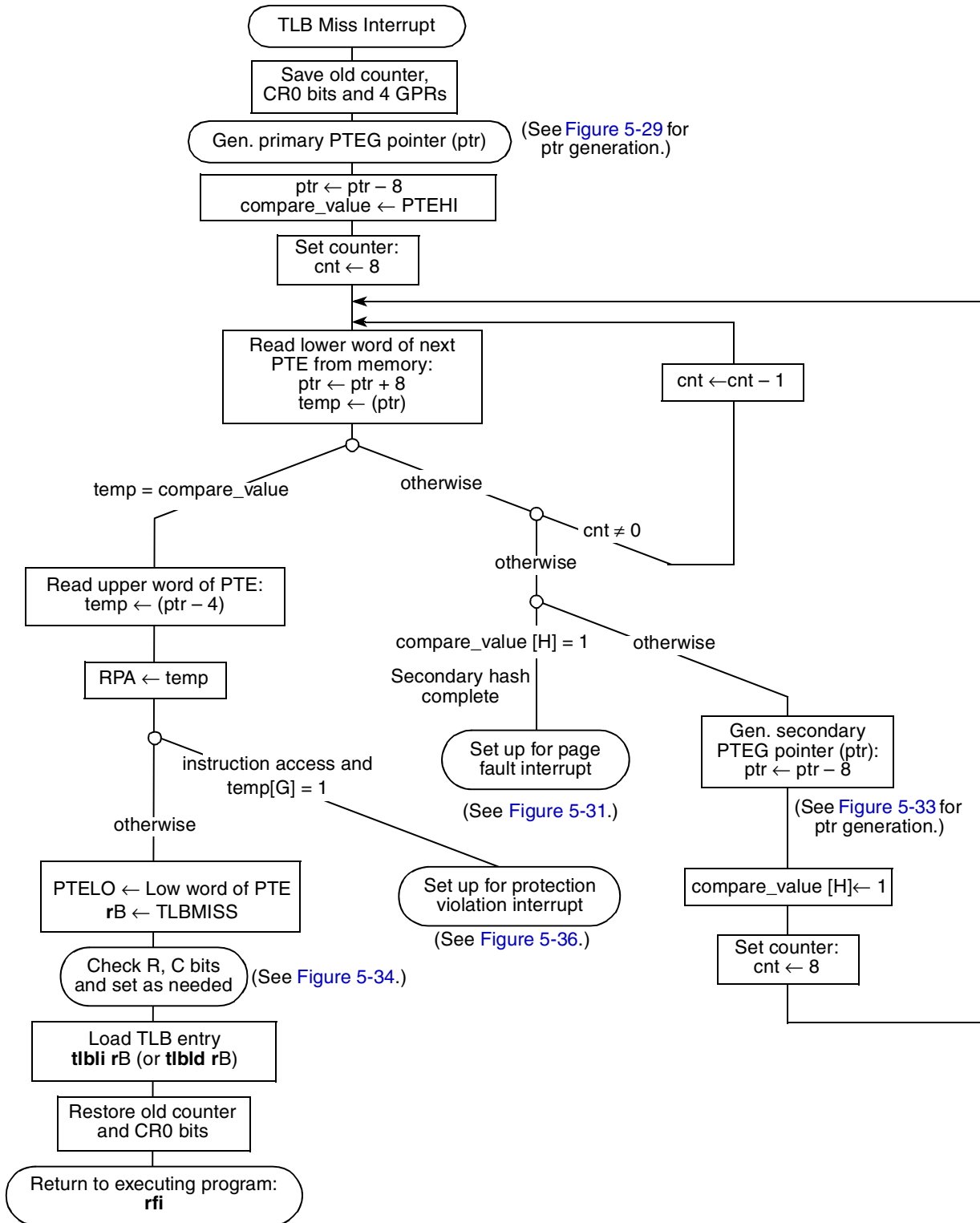


Figure 5-32. Flow for Example Software Table Search Operation

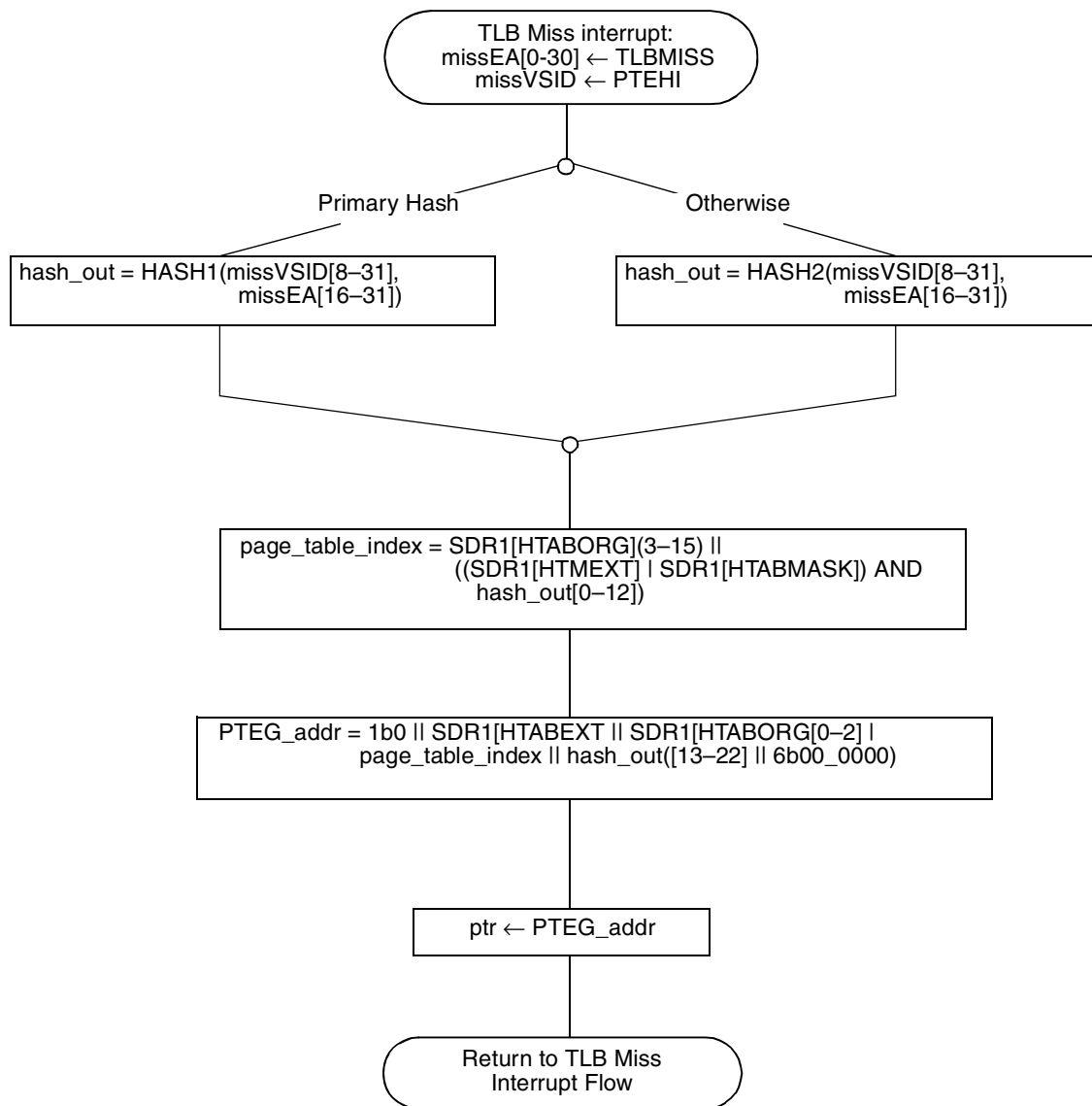


Figure 5-33. Flow for Generation of PTEG Address

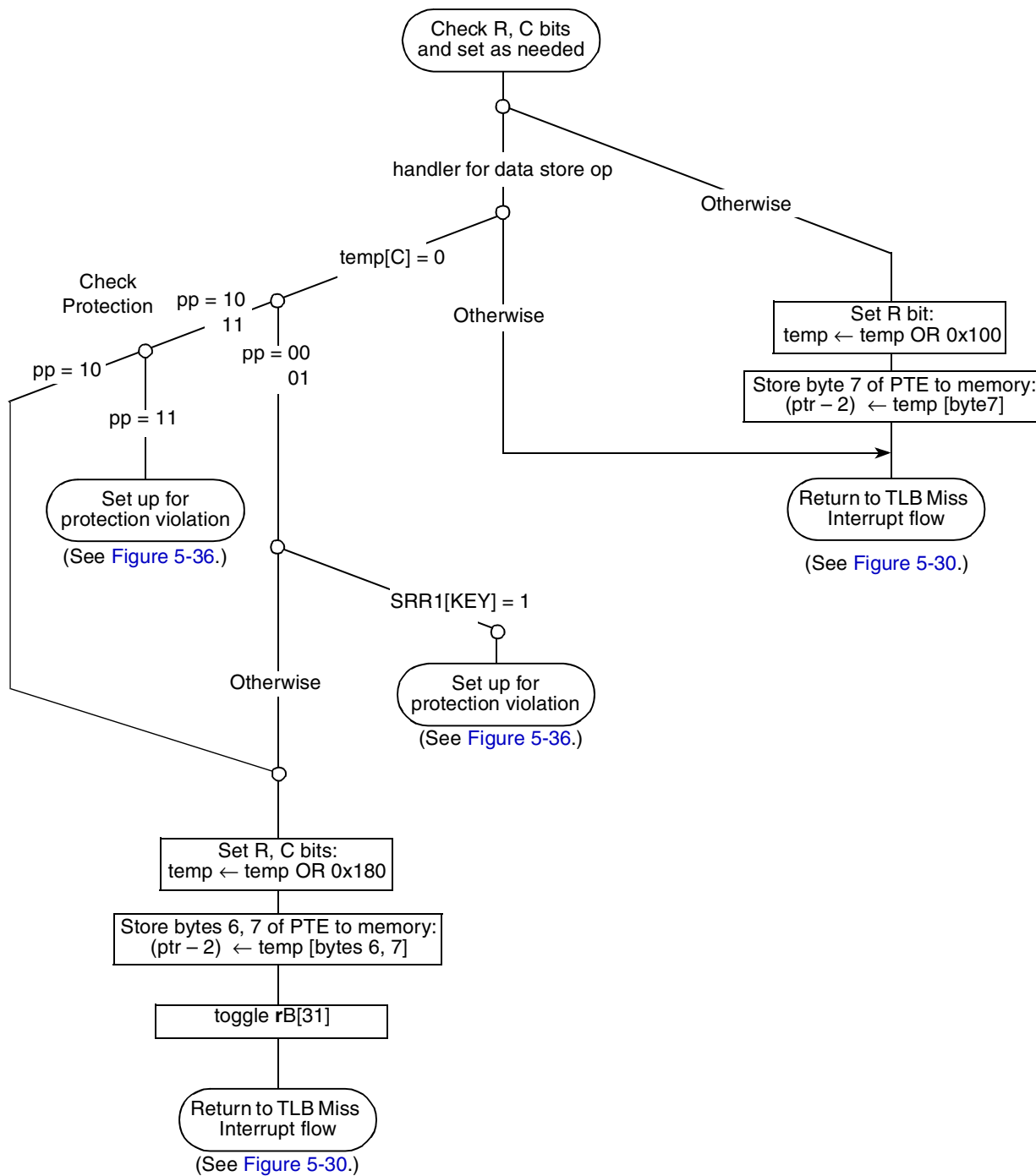


Figure 5-34. Check and Set R and C Bit Flow

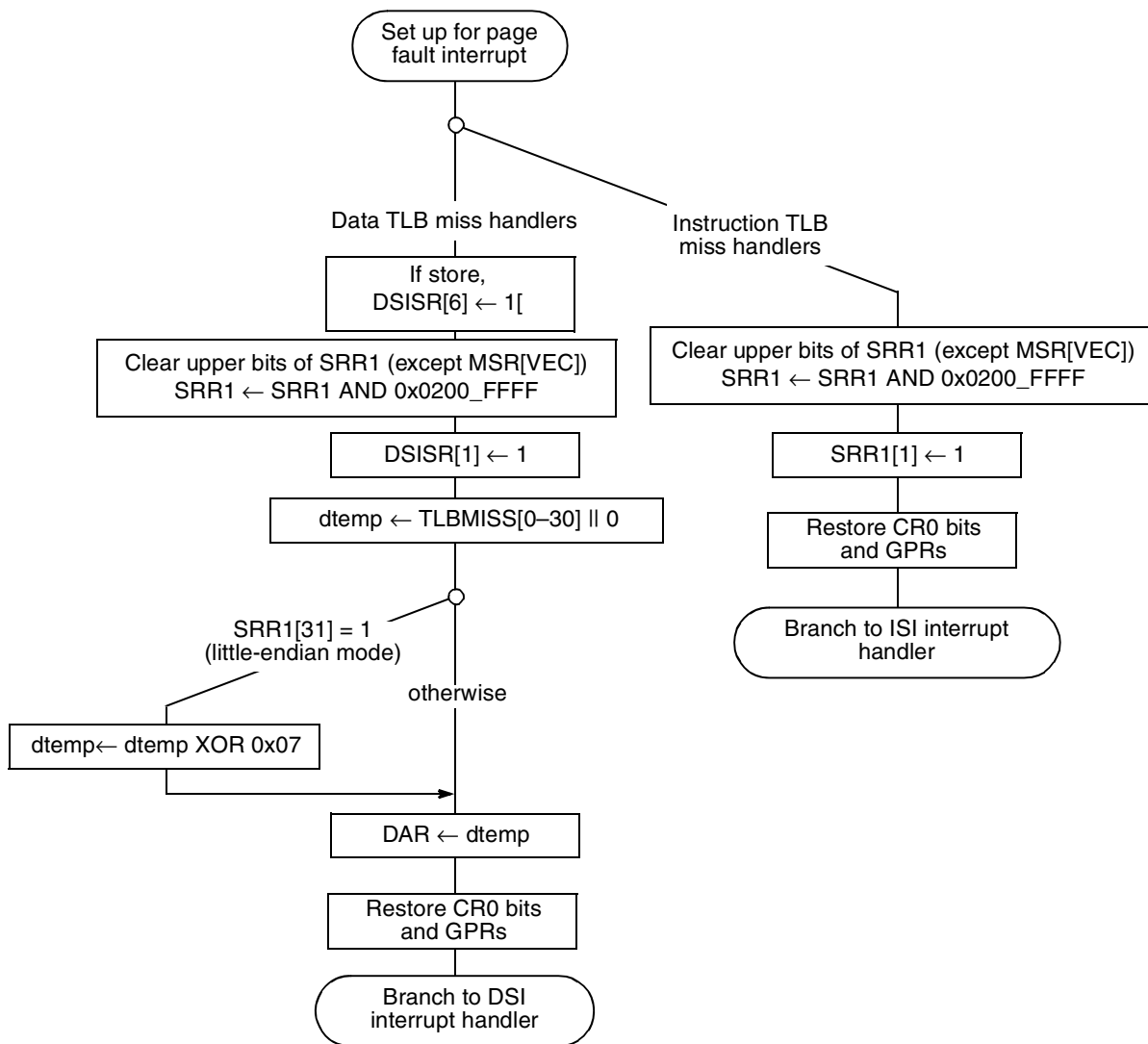


Figure 5-35. Page Fault Setup Flow

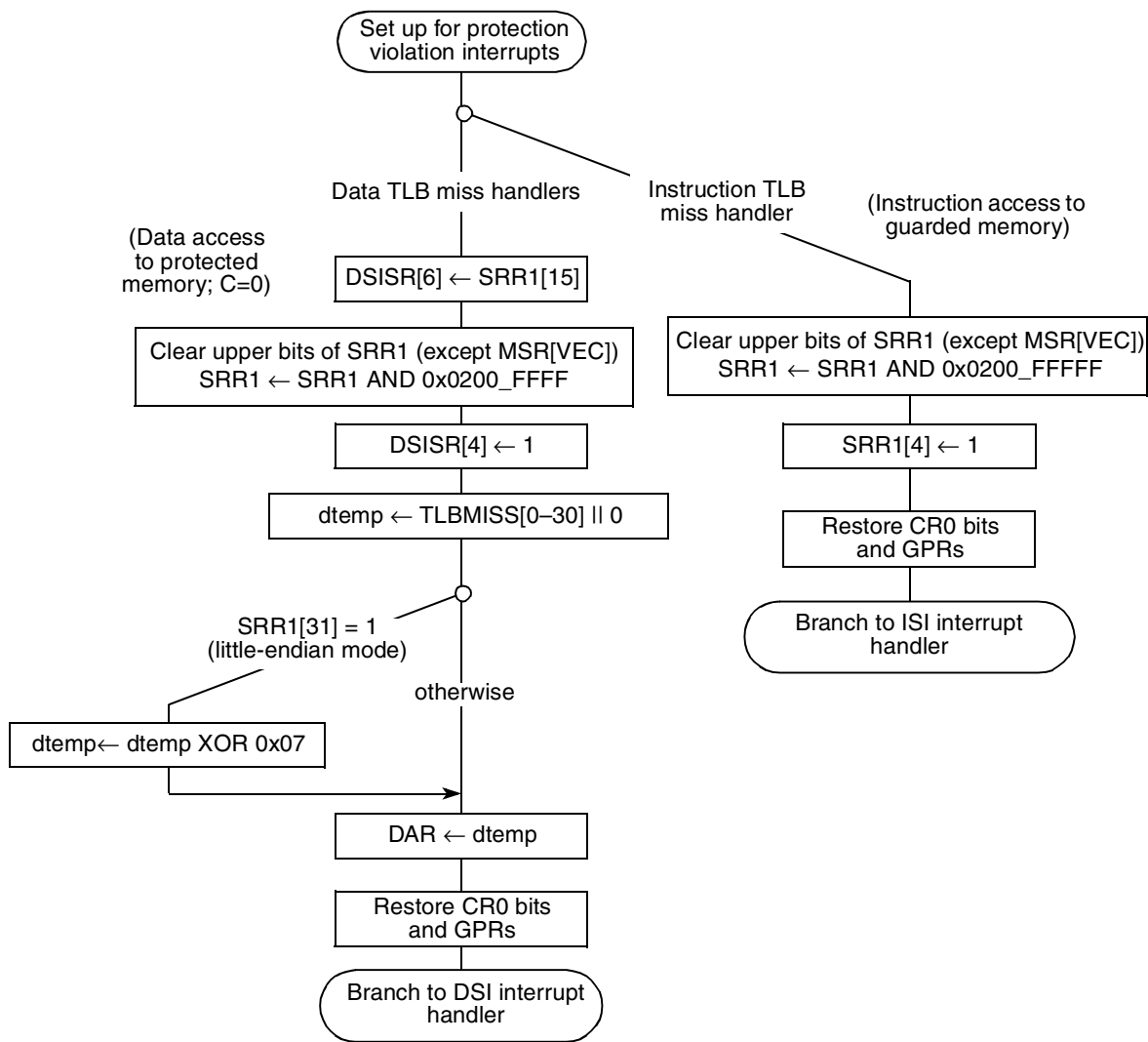


Figure 5-36. Setup for Protection Violation Interrupts

5.5.5.2.2 Code for Example Interrupt Handlers

This section provides some assembly language examples that implement the flow diagrams described above. Note that although these routines fit into a few cache lines, they are supplied only as a functional example; they could be further optimized for faster performance.

```

# TLB software load
#
# New Instructions:
#   tlbld   - write the dtlb with the ptehi and ptelo values
#   tlbli   - write the itlb with the ptehi and ptelo values
# New SPRs
#   tlbmiss - address of access that missed. Also contains LRU information
#   ptehi   - VSID of access that missed. Gets written to TLB on tlbld or tlbli
#   ptelo   - RPN value written to TLB on tlbld or tlbli
#
  
```

Memory Management

```

#
# there are three flows.
#   tlbDataMiss - tlb miss on data load
#   tlbCeq0     - tlb miss on data store or store with tlb changed bit == 0
#   tlbInstrMiss - tlb miss on instruction fetch
#+
# place labels for rel branches
#-
#.machine PPC_7450
# gpr r0..r3 are saved into SPR0-3
.set   r0, 0
.set   r1, 1
.set   r2, 2
.set   r3, 3
.set   tlbmiss, 980
.set   ptehi, 981
.set   ptelo, 982
.set   c0, 0
.set   dar, 19
.set   dsisr, 18
.set   srr0, 26
.set   srr1, 27
.set   sprg0, 272
.set   sprg1, 273
.set   sprg2, 274
.set   sprg3, 275
.
.csect tlbmiss[PR]
vec0:
.globl vec0

.org   vec0+0x300
vec300:
.org   vec0+0x400
vec400:
#+

# Interrupt vector serves as jump table
# Register usage:
#   Existing values of r0-r3 saved into sprg0-sprg3
# Note: It is assumed that the OS uses r31 as a pointer to the current top of stack.

.org   vec0+0x1000
# Instruction TLB Miss
    stwu r1,-4(r31) # store r1 to stack
    mflr r1         # save link register
    stwu r1,-4(r31) # store link register to stack
    bl   tlbInstrMiss # handler routine
    lwz  r1, 0(r31)  # link register from the stack
    addi r31,r31, 4  # pop stack
    mtlr r1         # restore the link register
    lwz  r1, 0(r31)  # r1 from the stack
    addi r31,r31, 4  # pop stack
    rfi

.org   vec0+0x1100
# Data TLB Miss
    stwu r1,-4(r31) # store r1 to stack

```



```

        mflr r1          # save link register
        stwu r1,-4(r31)  # store link register to stack
        bl   tlbDataMiss # handler routine
        lwz  r1, 0(r31)  # link register from the stack
        addi r31,r31, 4  # pop stack
        mtlr r1          # restore the link register
        lwz  r1, 0(r31)  # r1 from the stack
        addi r31,r31, 4  # pop stack
        rfi

.org    vec0+0x1200
# Data TLB Miss for Store
        stwu r1,-4(r31)  # store r1 to stack
        mflr r1          # save link register
        stwu r1,-4(r31)  # store link register to stack
        bl   tlbCeq0    # handler routine
        lwz  r1, 0(r31)  # link register from the stack
        addi r31,r31, 4  # pop stack
        mtlr r1          # restore the link register
        lwz  r1, 0(r31)  # r1 from the stack
        addi r31,r31, 4  # pop stack
        rfi

# Instruction TB miss flow
# Entry:
#     From Interrupt Vec = 1000
#     srr0 -> address of instruction that missed
#     srr1 -> 16:31 = saved MSR
#     tlbMiss-> ea that missed
#     tlbhi-> upper 32-bits of pte value
#     tlblo-> lower 32-bits of pte value
#
# Register usage:
#     r0-r3 used in the interrupt handler as follows
#     r0 is scratch pad
#     r1 is scratch pad
#     r2 is pointer to pteg
#     r3 is current compare value
#     r31 pointer to top of stack
# Note: It is assumed that the OS uses r31 as a pointer to the current top of #     stack.
.org    0x2000
tlbInstrMiss:
        mtsprsprg0, r0   # save r0 into sprg0
        mtsprsprg1, r1   # save r1 into sprg1
        mtsprsprg2, r2   # save r2 into sprg2
        mtsprsprg3, r3   # save r3 into sprg3
# Save CTR and CR on stack.
        mfctr0          # save counter
        stwu r0,-4(r31)  # store counter to stack
        mfcr r0         # save CR
        stwu r0,-4(r31)  # store CR to stack
        mfspr0, tlbMiss # EA of access that missed
        rlwinmr0,r0,20,16,31# Mask out lower 16 bits of EA
        mfspr1, ptehi   # VSID of access that missed
        rlwinmr1,r1,25,8,31# Mask out upper 23 bits of VSID
        xor  r1,r0,r1    # Primary HASH
        mfspr3,sdr1     # SDR1 value
        rlwinmr0,r3,10,13,31# align HTMEXT and HTABMASK fields

```

Memory Management

```

ori r0,r0,0x3ff # Mask out HTMEXT and HTABMASK
and r1,r0,r1 # and result
rlwinmr0,r3,26,13,21
or r1,r0,r1 # or result

# 32-bit PTEG Address generation into r2
andis.r2,r3,0xfe00
rlwimir2,r1,6,7,25

xor r1,r1,r1 # zero out reg.
addi r1, r0, 8 # load 8 for counter
mfctrr0 # save counter
mfspr3, ptehi # get first compare value
addi r2, r2, -8 # pre dec the pointer
im0:
mtctrr1 # load counter
im1:
lwzu r1,8(r2) # get next pte
cmp c0,r1,r3 # see if found pte
bdnzfeq,im1 # dec count br if cmp ne and if count not zero
bne instrSecHash # if not found set up second hash or exit
lwz r1,4(r2) # load tlb entry lower-word
andi.r3,r1,8 # check G-bit
bne doISIP # if guarded, take an ISI
ori r1,r1,0x100 # set referenced bit
mtsprptelo,r1 # put rpn into ptelo reg.
mfspr0, tlbmiss
tlblir0 # load the itlb
srwi r1,r1,8 # get byte 7 of pte
stb r1,6(r2) # update page table
# Restore application values
lwz r0,0(r31) # get counter value
addi r31,r31,4 # pop stack
mtctrr0 # restore counter
lwz r0,0(r31) # get CR value
addi r31,r31,4 # pop stack
mtcrf0xff,r0 # restore CR
mfspr0, sprg0 # restore old value of r0
mfspr1, sprg1 # restore old value of r1
mfspr2, sprg2 # restore old value of r2
mfspr3, sprg3 # restore old value of r3
blr # return to jump table

#+
# Register usage:
# r0 is saved counter
# r1 is junk
# r2 is pointer to pteg
# r3 is current compare value
#-
instrSecHash:
andi.r1,r3,0x0040 # see if we have done second hash
bne doISI # if so, go to ISI interrupt
mfspr0,tlbMiss # EA of access that missed
rlwinmr0,r0,20,16,31# Mask out lower 16 bits of EA
mfspr1,ptehi # VSID of access that missed
rlwinmr1,r1,25,8,31# Mask out uppder 23 bits of VSID

```

```

xor r1,r0,r1      # Primary HASH
mfspr3,sdr1      # SDR1 value
rlwinmr0,r3,10,13,31# align HTMEXT and HTABMASK fields
ori r0,r0,0x3ff  # Mask out HTMEXT and HTABMASK
and r1,r0,r1      # and result
rlwinmr0,r3,26,13,21
or r1,r0,r1       # or result
# 32-bit PTEG Address generation into r2
andis.r2,r3,0xfe00
rlwimir2,r1,6,7,25

ori r3,r3,0x0040 # change the compare value
addi r1, r0, 8    # load 8 for counter
addi r2, r2, -8   # pre dec for update on load
b im0            # try second hash

#+
# entry Not Found: synthesize an ISI interrupt
# guarded memory protection violation: synthesize an ISI interrupt
# Entry:
# r0 is saved counter
# r1 is junk
# r2 is pointer to pteg
# r3 is current compare value
#
doISIp:
mfspr3,srr1      # get srr1
andi.r2,r3,0xFFFF # clean upper srr1
addisr2,r2,0x0800 # or in srr<4> = 1 to flag prot violation
b isil

doISI:
mfspr3,srr1      # get srr1
andi r2,r3,0xFFFF # clean srr1
addisr2,r2,0x4000 # or in srr1<1> = 1 to flag pte not found

isil:
mtctrr0          # restore counter
mtpsrsrr1,r2     # set srr1
mtcrf0x80,r3     # restore CR0
mfspr0,sprg0     # restore old value of r0
mfspr1,sprg1     # restore old value of r1
mfspr2,sprg2     # restore old value of r2
mfspr3,sprg3     # restore old value of r3
b isiExc         # go to instr. access interrupt

+
# Data TLB miss flow
# Entry:
# From Interrupt Vec = 1100
# srr0 -> address of instruction that caused data tlb miss
# srr1 -> store 16:31 = saved MSR
# tlbMiss-> ea that missed
# tlbhi-> upper 32-bits of pte value
# tlblo-> lower 32-bits of pte value
#
# Register usage:
# r0 is scratch pad
# r1 is scratch pad
# r2 is pointer to pteg

```

Memory Management

```

#       r3 is current compare value
#       r3l pointer to top of stack
# Note: It is assumed that the OS has a stack for saving and restoring      #
#       application variables. r3l serves as a pointer to the current top    #
#       of stack.
#-
.csect  tlbmiss[PR]

tlbDataMiss:
    mtsprsprg0,r0      # save r0 into sprg0
    mtsprsprg1,r1      # save r1 into sprg1
    mtsprsprg2,r2      # save r2 into sprg2
    mtsprsprg3,r3      # save r3 into sprg3
# Save CTR and CR on stack.
    mfctrr0           # save counter
    stwu r0,-4(r31)   # store counter to stack
    mfcr r0           # save CR
    stwu r0,-4(r31)   # store CR to stack
    mfsprrr0,tlbMiss  # EA of access that missed
    rlwinmr0,r0,20,16,31# Mask out lower 16 bits of EA
    mfsprrr1,ptehi    # VSID of access that missed
    rlwinmr1,r1,25,8,31# Mask out upper 23 bits of VSID
    xor  r1,r0,r1     # Primary HASH
    mfsprrr3,sdr1     # SDR1 value
    rlwinmr0,r3,10,13,31# align HTMEXT and HTABMASK fields
    ori  r0,r0,0x3ff  # Mask out HTMEXT and HTABMASK
    and  r1,r0,r1     # and result
    rlwinmr0,r3,26,13,21
    or   r1,r0,r1     # or result
# 32-bit PTEG Address generation into r2
    andis.r2,r3,0xfe00
    rlwimir2,r1,6,7,25

    xor  r1,r1,r1     # zero out reg.
    addi r1,r1,8      # load 8 for counter
    mfsprrr3,ptehi   # get first compare value
    addi r2,r2,-8     # pre dec the pointer
dm0:
    mtctrr1          # load counter
dm1:
    lwzu r1,8(r2)     # get next pte
    cmp  c0,r1,r3     # see if found pte
    bdnzfeq,dm1      # dec count br if cmp ne and if count not zero
    bne  dataSecHash # if not found set up second hash or exit
    lwz  r1,4(r2)     # load tlb entry lower-word
    ori  r1,r1,0x100  # set referenced bit
    mtsprptelo,r1    # put rpn into ptelo reg.
    mfsprrr0,tlbmiss
    tlbldr0          # load the dtlb
    srwi r1,r1,8      # get byte 7 of pte
    stb  r1,6(r2)     # update page table
# Restore application values
    lwz  r0,0(r31)    # get counter value
    addi r31,r31,4    # pop stack
    mtctrr0          # restore counter
    lwz  r0,0(r31)    # get CR value
    addi r31,r31,4    # pop stack
    mtcrrf0xff,r0    # restore CR

```

```

    mfsprrr0,sprg0    # restore old value of r0
    mfsprrr1,sprg1    # restore old value of r1
    mfsprrr2,sprg2    # restore old value of r2
    mfsprrr3,sprg3    # restore old value of r3
    blr               # return to jump table

dataSecHash:
    andi.r1,r3,0x0040 # see if we have done second hash
    bne doDSI         # if so, go to DSI interrupt
    mfsprrr0,tlbMiss # EA of access that missed
    rlwinmr0,r0,20,16,31# Mask out lower 16 bits of EA
    mfsprrr1,pteHi    # VSID of access that missed
    rlwinmr1,r1,25,8,31# Mask out upper 23 bits of VSID
    xor r1,r0,r1      # Primary HASH
    mfsprrr3,sdr1     # SDR1 value
    rlwinmr0,r3,10,13,31# align HTMEXT and HTABMASK fields
    ori r0,r0,0x3ff   # Mask out HTMEXT and HTABMASK
    and r1,r0,r1      # and result
    rlwinmr0,r3,26,13,21
    or r1,r0,r1       # or result
# 32-bit PTEG Address generation into r2
    andisr2,r3,0xfe00
    rlwimir2,r1,6,7,25
    ori r3,r3,0x0040 # change the compare value?
    addi r1,r0,8      # load 8 for counter
    addi r2,r2,-8     # pre dec for update on load
    b dm0            # try second hash

# C=0 in dtlb and dtlb miss on store flow
# Entry:
# From Interrupt Vec = 1200
# srr0 -> address of store that caused the interrupt
# srr1 -> 16:31 = saved MSR
# tlbMiss-> ea that missed
# tlbHi-> upper 32-bits of pte value
# tlbLo-> lower 32-bits of pte value
#
# Register usage:
# r0 is saved counter
# r1 is junk
# r2 is pointer to pteg
# r3 is current compare value
# r31 pointer to top of stack
# Note: It is assumed that the OS has a stack for saving and restoring
# application variables. r31 serves as a pointer to the current # top of stack.
#-

.csect tlbmiss[PR]

tlbCeq0:
    mtspr sprg0,r0 # save r0 into sprg0
    mtspr sprg1,r1 # save r1 into sprg1
    mtspr sprg2,r2 # save r2 into sprg2
    mtspr sprg3,r3 # save r3 into sprg3
# Save CTR reg and CR on stack.
    mfctr r0 # save counter
    stwu r0,-4(r31)# store counter to stack
    
```

Memory Management

```

mfcr    r0          # save CR
stwu   r0,-4(r31)# store CR to stack
mfspr  r0,tlbMiss# EA of access that missed
rlwinm r0,r0,20,16,31# Mask out lower 16 bits of EA
mfspr  r1,pteHi    # VSID of access that missed
rlwinm r1,r1,25,8,31# Mask out upper 23 bits of VSID
xor    r1,r0,r1    # Primary HASH
mfspr  r3,sdr1    # SDR1 value
rlwinm r0,r3,10,13,31# align HTMEMT and HTABMASK fields
ori    r0,r0,0x3ff# Mask out HTMEMT and HTABMASK
and    r1,r0,r1    # and result
rlwinm r0,r3,26,13,21
or     r1,r0,r1    # or result
# 32-bit PTEG Address generation into r2
andis. r2,r3,0xfe00
rlwimi r2,r1,6,7,25

xor    r1,r1,r1    # zero out reg.
addi   r1,r1,8     # load 8 for counter
mfspr  r3,pteHi    # get first compare value
addi   r2,r2,-8    # pre dec the pointer
ceq0:
mtctr  r1          # load counter
ceq1:
lwzu   r1,8(r2)    # get next pte
cmp    c0,r1,r3    # see if found pte
bdnzf  eq,ceq1    # dec count br if cmp ne and if count not zero
bne    cEq0SecHash # if not found set up second hash or exit
lwz    r1,4(r2)    # load tlb entry lower-word
andi.  r3,r1,0x80  # check the C-bit
beq    cEq0ChkProt # if (C==0) go check protection modes
ceq2:
mfspr  ptelo,r1    # put rpn into ptelo reg.
mfspr  r0,tlbMiss
xori   r0,r0,0x01 # toggles lru bit
tlbld  r0          # load the dtlb
# Restore application values
lwz    r0,0(r31)   # get counter value
addi   r31,r31,4   # pop stack
mtctr  r0          # restore counter
lwz    r0,0(r31)   # get CR value
addi   r31,r31,4   # pop stack
mtrcrf 0xff,r0    # restore CR
mfspr  r0,sprg0    # restore old value of r0
mfspr  r1,sprg1    # restore old value of r1
mfspr  r2,sprg2    # restore old value of r2
mfspr  r3,sprg3    # restore old value of r3
blr    # return to jump table
#+
# Register usage:
#    r0 is saved counter
#    r1 is junk
#    r2 is pointer to pteg
#    r3 is current compare value
#-
```

```

ceq0SecHash:
    andi    r1,r3,0x0040 # see if we have done second hash
    bne     doDSI        # if so, go to DSI interrupt
    andi    r1,r3,0x004 # see if we have done second hash
    bne     doDSI        # if so, go to DSI interrupt
    mfspr   r0,tlbMiss   # EA of access that missed
    rlwinm  r0,r0,20,16,3 # Mask out lower 16 bits of EA
    mfspr   r1,pteh      # VSID of access that missed
    rlwinm  r1,r1,25,8,31 # Mask out upper 23 bits of VSID
    xor     r1,r0,r1     # Primary HASH
    mfspr   r3,sdr1     # SDR1 value
    rlwinm  r0,r3,10,13,3 # align HTMEXT and HTABMASK fields
    ori     r0,r0,0x3f   # Mask out HTMEXT and HTABMASK
    and     r1,r0,r1     # and result
    rlwinm  r0,r3,26,13,21
    or      r1,r0,r1     # or result
# 32-bit PTEG Address generation into r2
    andis.  r2,r3,0xfe00
    rlwimi  r2,r1,6,7,25

    ori     r3,r3,0x004 # change the compare value
    addi    r1,r0,8     # load 8 for counter
    addi    r2,r2,-8    # pre dec for update on load
    b       ceq0       # try second hash

#+
# entry found and PTE(c-bit==0):
# (check protection before setting PTE(c-bit))
# Register usage:
#     r0 is saved counter
#     r1 is PTE entry
#     r2 is pointer to pteg
#     r3 is trashed
#-
ceq0ChkProt:
    rlwinm  r3,r1,30,0,1 # test PP
    bge-    chk0         # if (PP==00 or PP==01) goto chk0:
    andi    r3,r1,1      # test PP[0]
    beq+    chk2         # return if PP[0]==0
    b       doDSIp      # else DSIP

chk0:
    mfspr   r3,srr1     # get old msr
    andis   r3,r3,0x0008 # test the KEY bit (SRR0-bit 12)
    b       doDSIp     # else DSIP

chk2:
    ori     r1,r1,0x180 # set referenced and changed bit
    sth     r1,6(r2)    # update page table
    b       ceq2       # and back we go
    #

#+
# entry Not Found: synthesize a DSI interrupt
# Entry:
#     r0 is saved counter
#     r1 is junk
#     r2 is pointer to pteg
    
```

Memory Management

```

#       r3 is current compare value
#
doDSI:
    mfspr    r3,srr1      # get srr1
    rlwinm   r1,r3,9,6,6 # get srr1<flag> to bit 6 for load/store, zero rest
    addis    r1,r1,0x4000 # or in dsisr<1> = 1 to flag pte not found
    b        dsil

doDSIp:
    mfspr    r3, srr1     # get srr1
    rlwinm   r1, r3,9,6,6 # get srr1<flag> to bit 6 for load/store, zero rest
    addis    r1, r1, 0x0800# or in dsisr<4> = 1 to flag prot violation

dsil:
    mtctr    r0           # restore counter
    andis.   r2,r3, 0x0200 # Keep the AltiVec Avail bit in r2
    andi     r3,r3,0xFFFF # Zero out the upper bits of SRR1
    or       r2, r2, r3    # OR back in the lower bits into r2
    mtspr    srr1, r2     # set srr1
    mtspr    dsisr, r1    # load the dsisr
    mfspr    r1, tlbmiss  # get miss address
    rlwinm   r1,r1,0,0,30 # Clear the LRU bit
    rlwinm.  r2,r2,0,31,31 # test LE bit
    beq      dsil2       # if little endian then:
    xori     r1,r1,0x07   # de-mung the data address

dsi2:
    mtspr    dar,r1      # put in dar
    mtcrf   0x80,r3     # restore CR0?
    mfspr    r0,sprg0   # restore old value of r0
    mfspr    r1,sprg1   # restore old value of r1
    mfspr    r2,sprg2   # restore old value of r2
    mfspr    r3,sprg3   # restore old value of r3
    b        dsiExc     # branch to DSI interrupt

```


Chapter 6

Instruction Timing

This chapter describes how the e600 core performs operations defined by instructions and how it reports the results of instruction execution. It gives detailed descriptions of how the e600 execution units work and how these units interact with other parts of the processor, such as the instruction fetching mechanism, register files, and caches. It gives examples of instruction sequences, showing potential bottlenecks and how to minimize their effects. Finally, it includes tables that identify the unit that executes each instruction implemented on the e600 core, the latency for each instruction, and other information useful to assembly language programmers.

AltiVec Technology and Instruction Timing

The AltiVec functionality in the e600 core affects instruction timing in the following ways:

- Execution units are provided for vector computations:
 - Vector permute unit (VPU). See [Section 6.4.5.1.1, “AltiVec Permute Unit \(VPU\) Execution Timing.”](#)
 - Short-latency vector integer unit 1 (VIU1). See [Section 6.4.5.1.2, “Vector Simple Integer Unit \(VIU1\) Execution Timing.”](#)
 - Long-latency vector complex integer unit (VIU2). See [Section 6.4.5.1.3, “Vector Complex Integer Unit \(VIU2\) Execution Timing.”](#)
 - Vector floating-point unit (VFPU). See [Section 6.4.5.1.4, “Vector Floating-Point Unit \(VFPU\) Execution Timing.”](#)
- The AltiVec technology defines data streaming instructions that allow automated loading of data for non-speculative accesses. These instructions can be identified as either static (likely to be reused) or transient (unlikely to be reused). See [Chapter 7, “AltiVec Technology Implementation.”](#)
- The AltiVec technology defines the difference between the instructions `lvxl` and `stvx1` with other AltiVec load and store instructions. See [Section 6.4.4.3.1, “LRU Instructions.”](#)

6.1 Terminology and Conventions

This section provides an alphabetical glossary of terms used in this chapter. These definitions offer a review of commonly used terms and point out specific ways these terms are used in this chapter.

NOTE

Many of these definitions differ slightly from those used to describe previous processors and cores that implement the PowerPC architecture, in particular with respect to dispatch, issue, finishing, retirement, and write back, so please read this glossary carefully.

- Branch prediction—The process of guessing the direction or target of a branch. Branch direction prediction involves guessing whether a branch will be taken. Target prediction involves guessing the target address of a **bclr** branch. The PowerPC architecture defines a means for static branch prediction as part of the instruction encoding.
- Branch resolution—The determination of whether a branch prediction was correct or not. If the prediction is correct, the instructions following the predicted branch that may have been speculatively executed can complete (*see* completion). If the prediction is incorrect, instructions on the mispredicted path and any results of speculative execution are purged from the pipeline and fetching continues from the correct path.
- Complete—An instruction is in the complete stage after it executes and makes its results available for the next instruction (*see* finish). At the end of the complete stage, the instruction is retired from the completion queue (CQ). When an instruction completes, it is guaranteed that this instruction and all previous instructions can cause no interrupts.
- Dispatch—The dispatch stage decodes instructions supplied by the instruction queue, renames any source/target operands, determines to which issue queue each non-branch instruction is dispatched, and determines whether the required space is available in both that issue queue and the completion queue.
- Fall-through folding (branch fall-through)—Removal of a not-taken branch. On the e600 core, not-taken branch instructions that do not update LR or CTR can be removed from the instruction stream if the branch instruction is in IQ3–IQ7 the cycle after execution.
- Fetch—The process of bringing instructions from memory (such as a cache or system memory) into the instruction queue.
- Finish—An executed instruction finishes by updating the completion queue that execution is complete and results have been made available to subsequent instructions. For most execution units, finishing occurs at the end of the last cycle of execution; however, FPU, IU2, and VIU2 instructions finish at the end of a single-cycle finish stage after the last cycle of execution.
- Folding (branch folding)—The replacement with target instructions of a branch instruction and any instructions along the not-taken path when a branch is either taken or predicted as taken.
- Issue—The pipeline stage responsible for reading source operands from rename registers and register files. This stage also assigns and routes instructions to the proper execution unit.
- Latency—The number of clock cycles necessary to execute an instruction and make the results of that execution available to subsequent instructions.
- Pipeline—In the context of instruction timing, the term ‘pipeline’ refers to the interconnection of the stages. The events necessary to process an instruction are broken into several cycle-length tasks to allow work to be performed on several instructions simultaneously—analogue to an assembly line. As an instruction is processed, it passes from one stage to the next. When it does, the stage becomes available for the next instruction.

Although an individual instruction can take many cycles to make results available (*see* latency), pipelining makes it possible to overlap processing so that the throughput (number of instructions processed per cycle) is greater than if pipelining were not implemented.

- Program order—The order of instructions in an executing program. More specifically, this term is used to refer to the original order in which program instructions are fetched into the instruction queue from the cache.
- Rename registers—Temporary buffers for holding results of instructions that have finished execution but have not completed
- Reservation station—A buffer between the issue and execute stages that allows instructions to be issued even though the results of other instructions on which the issued instruction may depend are not available.
- Retirement—Removal of a completed instruction from the CQ.
- Speculative instruction—Any instruction which is currently behind an older branch that has not been resolved yet.
- Stage—Used in two different senses, depending on whether the pipeline is being discussed as a physical entity or a sequence of events. In the latter case, a stage is an element in the pipeline during which certain actions are performed, such as decoding the instruction, performing an arithmetic operation, or writing back the results. Typically, the latency of a stage is one processor clock cycle. Some events, such as dispatch, write-back, and completion, happen instantaneously and may be thought to occur at the end of a stage.

An instruction can spend multiple cycles in one stage. An integer multiply, for example, takes multiple cycles in the execute stage. When this occurs, subsequent instructions may stall.

An instruction can also occupy more than one stage simultaneously, especially in the sense that a stage can be seen as a physical resource—for example, when instructions are dispatched they are assigned a place in the CQ at the same time they are passed to the issue queues.

- Stall—An occurrence when an instruction cannot proceed to the next stage.
- Superscalar—A superscalar processor is one that can issue multiple instructions concurrently from a conventional linear instruction stream. In a superscalar implementation, multiple instructions can be in the execute stage at the same time.
- Throughput—The number of instructions that are processed per cycle. For example, a series of **muli** instructions have a throughput of one instruction per clock cycle.
- Write-back—Write-back (in the context of instruction handling) occurs when a result is written into the architecture-defined registers (typically the GPRs, FPRs, and VRs). On the e600 core, write back occurs in the clock cycle after the completion stage. Results in the write-back buffer cannot be flushed. If an interrupt occurs, results from previous instructions must write back before the interrupt is taken.

6.2 Instruction Timing Overview

The design of the e600 core minimizes the number of clock cycles it takes to fetch, decode, dispatch, issue, and execute instructions and to make the results available for a subsequent instruction. Some instructions, such as loads and stores, access memory and require additional clock cycles between the execute phase and the write-back phase. These latencies vary depending on whether the access is to cacheable or noncacheable memory, whether it hits in the L1 or L2 cache, whether the cache access generates a write-back to memory, whether the access causes a snoop hit from another device that generates additional activity, and other conditions that affect memory accesses.

To improve throughput, the e600 core implements pipelining, superscalar instruction issue, branch folding, removal of fall-through branches, three-level speculative branch handling, and multiple execution units that operate independently and in parallel.

As an instruction passes from stage to stage, the subsequent instruction can follow through the stages as the former instruction vacates them, allowing several instructions to be processed simultaneously. Although it may take several cycles for an instruction to pass through all the stages, when the pipeline is full, one instruction can complete on every clock cycle. [Figure 6-1](#) represents a generic four-stage pipelined execution unit, which when filled has a throughput of one instruction per clock cycle. E0–E3 show execution stages 1 through 4.

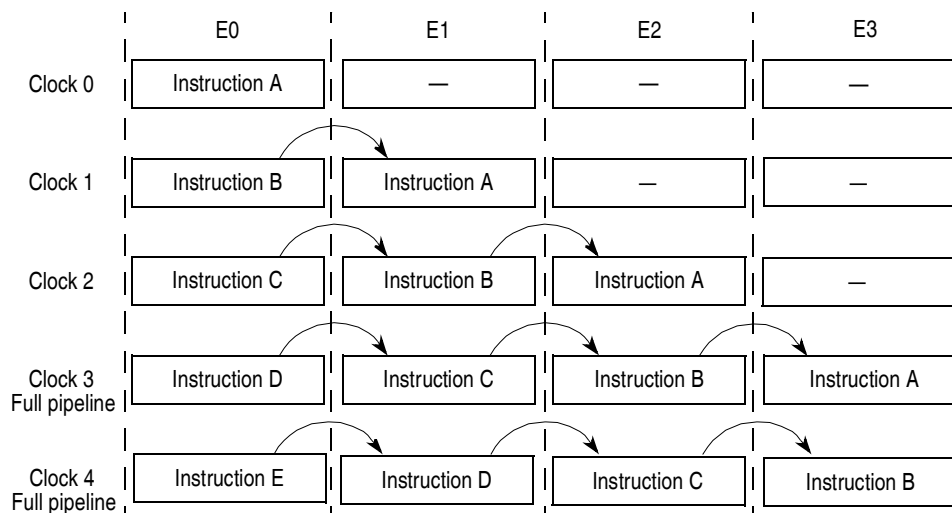


Figure 6-1. Pipelined Execution Unit

[Figure 6-2](#) shows the entire path that instructions take through the fetch1, fetch2, decode/dispatch, execute, issue, complete, and write-back stages, which is considered the e600 core’s master pipeline. The FPU, LSU, IU2, VIU2, VFPU, and VPU are also multiple-stage pipelines.

The e600 core contains the following execution units:

- Branch processing unit (BPU)
- Three single-cycle IUs (IU1a, IU1b, IU1c)—executes all integer (fixed-point) instructions except multiply, divide, and move to/from special-purpose register instructions. Note that all IU1 instructions execute in 1 cycle, except for some instructions like `tw[i]` and `srav[i][.]`, which take 2 cycles. See [Table 6-5](#) for details.
- Multiple-cycle IU (IU2)—executes miscellaneous instructions including the CR logical operations, integer multiplication and division instructions, and move to/from special-purpose register instructions
- 64-bit floating-point unit (FPU)
- Load/store unit (LSU)
- The AltiVec unit contains the following four independent execution units for vector computations, whose latencies are shown in [Table 6-8](#):
 - Vector permute unit (VPU)
 - Vector simple integer unit (VIU1)

- Vector complex integer unit (VIU2)
- Vector floating-point unit (VFPU)

A maximum of two AltiVec instructions can be issued out-of-order to any combination of AltiVec execution units per clock cycle from the bottom two VIQ entries (VIQ1–VIQ0). This means an instruction in VIQ1 destined for VIU1 does not have to wait for an instruction in VIQ0 that is stalled behind an instruction waiting for operand availability. Moreover, the VIU2, VFPU, and VPU are pipelined, so they can operate on multiple instructions.

The e600 core can complete as many as three instructions on each clock cycle. In general, the e600 core processes instructions in seven stages—fetch1, fetch2, decode/dispatch, issue, execute, complete, and write-back as shown in Figure 6-2. Note that the pipeline example in Figure 6-1 is similar to the four-stage VFPU pipeline in Figure 6-2.

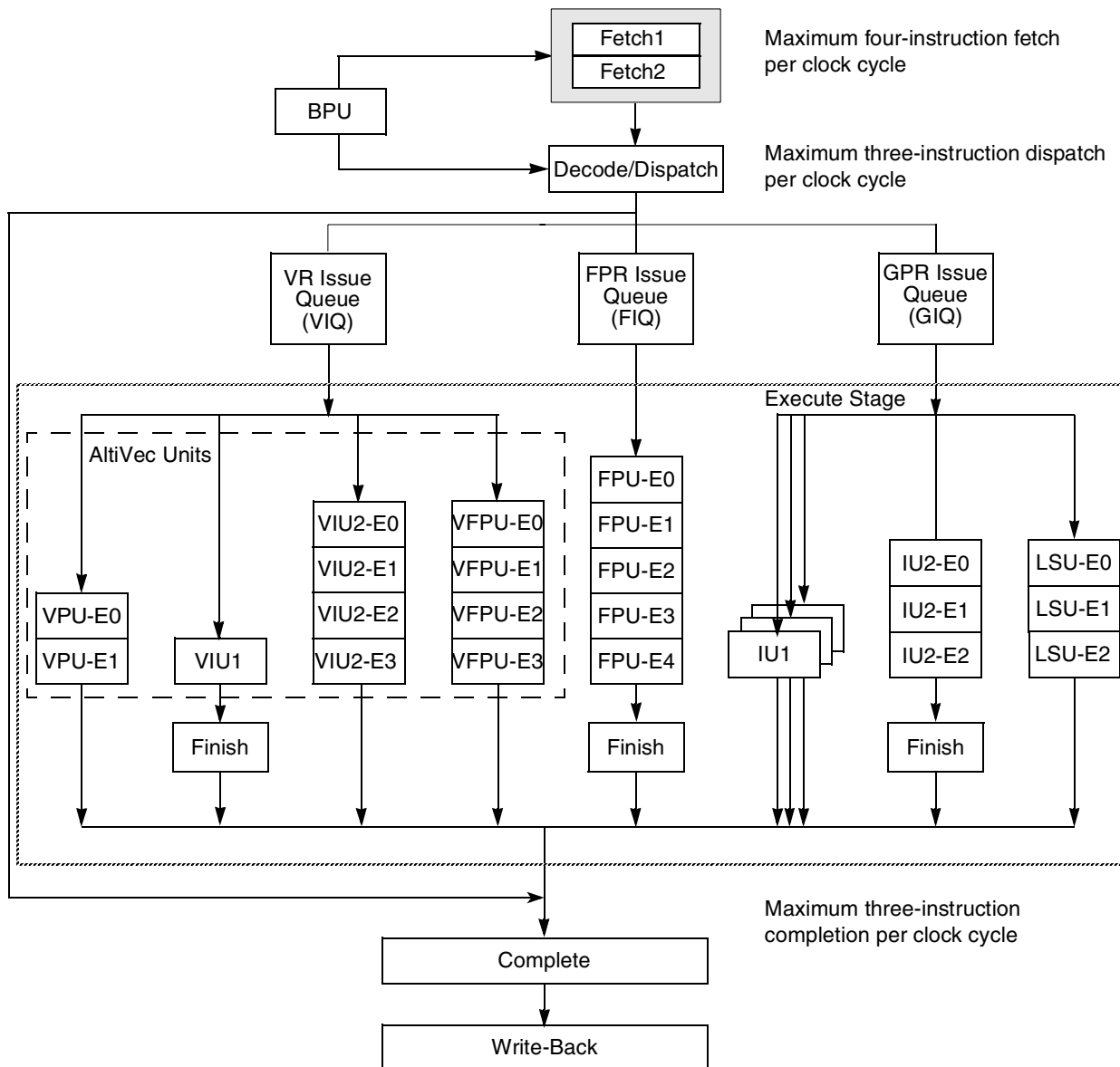


Figure 6-2. Superscalar/Pipeline Diagram

The instruction pipeline stages are described as follows:

- Instruction fetch—Includes the clock cycles necessary to request an instruction and the time the memory system takes to respond to the request. Instructions retrieved are latched into the instruction queue (IQ) for subsequent consideration by the dispatcher.
Instruction fetch timing depends on many variables, such as whether an instruction is in the branch target instruction cache (BTIC), the instruction cache, or the L2 cache. Those factors increase when it is necessary to fetch instructions from system memory and include the core-to-MPX bus clock ratio, the amount of MPX bus traffic, and whether any cache coherency operations are required. Because there are so many variables, unless otherwise specified, the instruction timing examples in this chapter assume optimal performance and show the portion of the fetch stage in which the instruction is in the instruction queue. The fetch1 and fetch2 stages are primarily involved in retrieving instructions.
- The decode/dispatch stage fully decodes each instruction; most instructions are dispatched to the issue queues (branch, **isync**, **rfi**, and **sc** instructions do not go to issue queues).
- The three issue queues FIQ, VIQ, and GIQ can accept as many as one, two, and three instructions, respectively, in a cycle. Instruction dispatch requires the following:
 - Instructions can be dispatched only from the three lowest IQ entries—IQ0, IQ1, and IQ2.
 - A maximum of three instructions can be dispatched to the issue queues per clock cycle.
 - Space must be available in the CQ for an instruction to dispatch (this includes instructions that are assigned a space in the CQ but not in an issue queue).

In this chapter, dispatch is treated as an event at the end of the fetch stage. Dispatch dependencies are described in [Section 6.7.2, “Dispatch Unit Resource Requirements.”](#)

The issue stage reads source operands from rename registers and register files and determines when instructions are latched into the execution unit reservation stations. The GIQ, FIQ, and VIQ (AltiVec) issue queues have the following similarities:

- Operand lookup in the GPRs, FPRs, and VRs, and their rename registers
- Issue queues issue instructions to the proper execution units.
- Each issue queue holds twice as many instructions as can be dispatched to it in 1 cycle; the GIQ has six entries, the VIQ has four, and the FIQ has two.

The three issue queues are described as follows:

- The GIQ accepts as many as three instructions from the dispatch unit each cycle. IU1, IU2, and all LSU instructions (including floating-point and AltiVec loads and stores) are dispatched to the GIQ.
- Instructions can be issued out-of-order from the bottom three GIQ entries (GIQ2–GIQ0). An instruction in GIQ1 destined for an IU1 does not have to wait for an instruction in GIQ0 that is stalled behind a long-latency integer divide instruction in the IU2.
- The VIQ accepts as many as two instructions from the dispatch unit each cycle. All AltiVec instructions (other than load, store, and vector touch instructions) are dispatched to the VIQ. As many as two instructions can be issued out-of-order to any combination of AltiVec execution units per clock cycle from the bottom two VIQ entries (VIQ1–VIQ0). An instruction in VIQ1 destined for VIU1 does not have to wait for an instruction in VIQ0 that is stalled

behind an instruction waiting for operand availability. Moreover, the VIU2, VFPU, and VPU are pipelined, so they can operate on multiple instructions.

- The FIQ can accept one instruction from the dispatch unit per cycle. It looks at the first instruction in its queue and determines if the instruction can be issued to the FPU in this cycle.
- The execute stage accepts instructions from its issue queue when the appropriate reservation stations are not busy. In this stage, the operands assigned to the execution stage from the issue stage are latched.

The execution unit executes the instruction (perhaps over multiple cycles), writes results on its result bus, and notifies the CQ when the instruction finishes. The execution unit reports any interrupts to the completion stage. Instruction-generated interrupts are not taken until the interrupting instruction is next to retire.

Most integer instructions have a 1-cycle latency, so results of these instructions are available 1 clock cycle after an instruction enters the execution unit. The FPU, LSU, IU2, VIU2, VFPU, and VPU units are pipelined, as shown in [Figure 6-3](#).

Note that AltiVec computational instructions are executed in the four independent, pipelined AltiVec execution units. The VPU has a two-stage pipeline, the VIU1 has a one-stage pipeline, and the VIU2 and VFPU have four-stage pipelines. As many as 10 AltiVec instructions can execute concurrently.

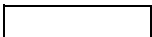

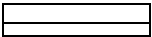
- The complete and write-back stages maintain the correct architectural machine state and commit results to the architecture-defined registers in the proper order. If completion logic detects an instruction containing an interrupt status, all following instructions are cancelled, their execution results in rename buffers are discarded, and the correct instruction stream is fetched.

The complete stage ends when the instruction is retired. Three instructions can be retired per clock cycle. If no dependencies exist, as many as three instructions are retired in program order.





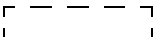
[Section 6.7.4, “Completion Unit Resource Requirements,”](#) describes completion dependencies.

The write-back stage occurs in the clock cycle after the instruction is retired.

Notation conventions used in the instruction timing examples in [Figure 6-8](#) through [Figure 6-16](#) are as follows:

-  Fetch—Instructions are fetched from memory and placed in the 12-entry IQ. The latency associated with accessing an instruction depends on whether the instruction is in the BTIC, the caches on the core, or system memory (in which case latency is further affected by bus traffic, bus clock speed, and address translation issues). Therefore, in the examples in this chapter, the diagrams and fetch stage shown are for the common case of instructions hitting in the instruction cache.
-  Branch execute—The operations specified by a branch instruction are being performed by the BPU. In some cases, the branch direction or target may be predicted. The white stripe is a reminder that the branch instruction occupies an entry in the IQ.
-  Dispatch—As many as three eligible instructions move, in order, from the IQ0–IQ2 to the appropriate issue queue. Note that branch, **isync**, **rfi**, and **sc** instructions do not go to issue queues. At the same time, the instruction is assigned an entry in the completion queue.

Instruction Timing

- 
Issue—Instructions are dispatched to issue queues from the instruction queue entries. At the end of the issue stage, instructions and their operands are latched into execution unit reservation stations. The black stripe is a reminder that the instruction occupies an entry in the CQ, described in [Figure 6-3](#).
- 
Execute—The operations specified by an instruction are being performed by the appropriate execution unit. The black stripe is a reminder that the instruction occupies an entry in the CQ, described in [Figure 6-3](#).
- 
Finish (FPU, IU2, and VIU1 only)—The single-cycle finish stage is required for all FPU, IU2, and VIU1 instructions to notify the completion logic that an instruction has executed and its results have been made available to rename registers.
- 
Complete—Execution has finished. When all completion requirements are met, the instruction is retired from the CQ. The results are written back to architecture-defined registers in the clock cycle after retirement.
- 
Write back—The instruction has retired and its results are written back to the architecture-defined registers.

The following events are associated with the stages described above:

- Dispatch—An instruction is dispatched to the appropriate issue queue at the end of the dispatch stage. At dispatch, the instruction passes to the issue pipeline stage by taking a place in the completion queue and in one of the three issue queues.
- Issue—The issue stage ends when the instruction is issued to the appropriate execution unit.
- Finish—An instruction finishes when the CQ is signalled that execution results are available to subsequent instructions. Architecture-defined registers are not updated until the instruction is retired. For FPU, IU2, and VIU2, finishing occurs at the end of a separate, one-cycle stage after the final execution stage.
- Retire—An instruction is retired when it has updated architecture-defined registers with its results and is removed from the completion queue.
- Write back—The results of a retired instruction are written back to the architecture-defined register.

[Figure 6-3](#) shows the relationships between stages and events.

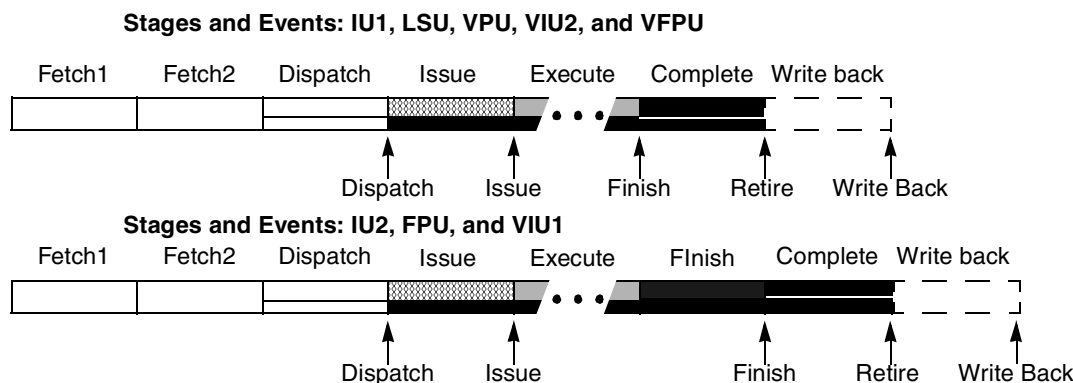
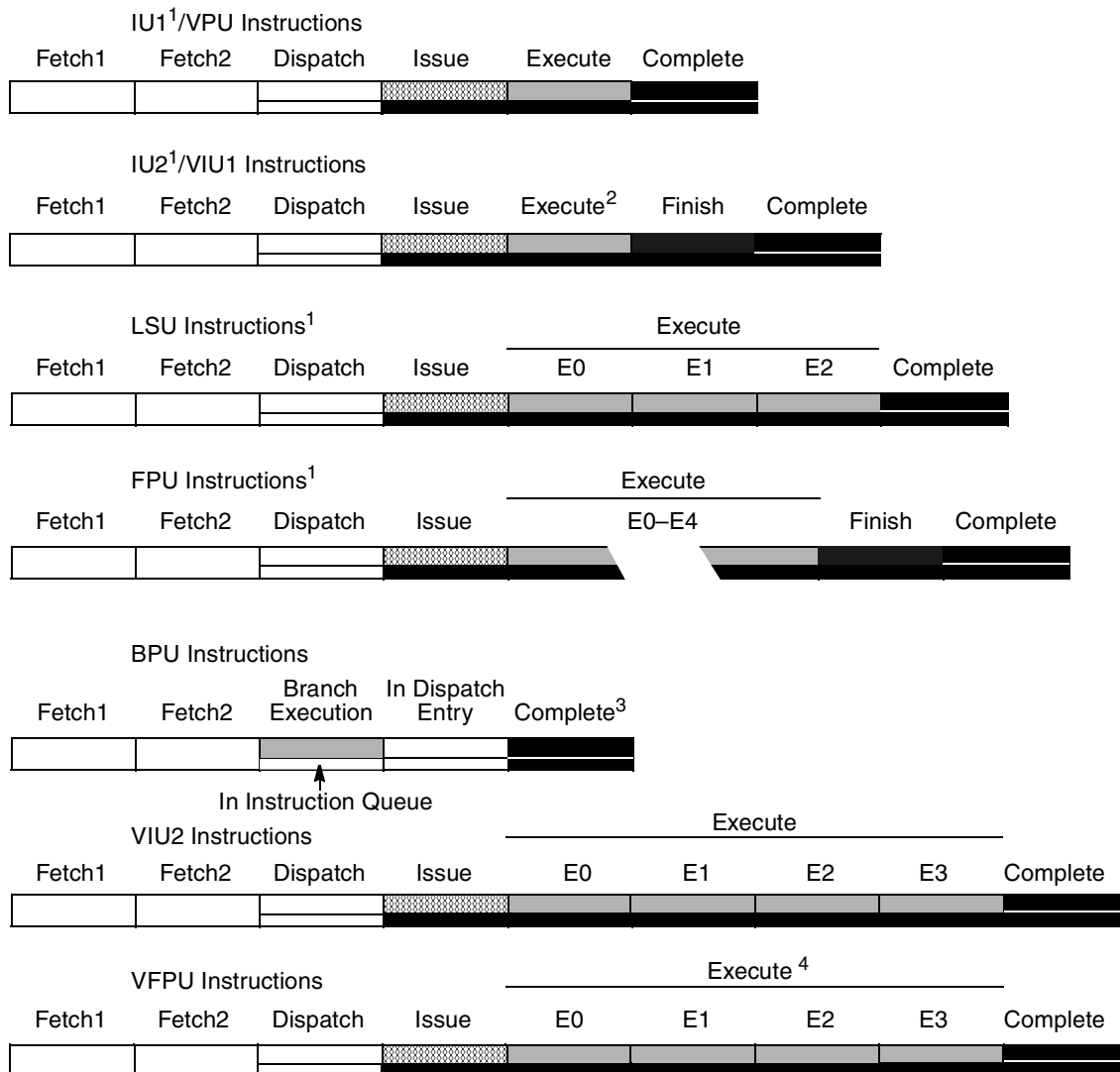


Figure 6-3. Stages and Events

Figure 6-4 shows the stages of e600 core execution units.



¹ Execution-serialized instructions require additional execution cycles (not shown).

² Several integer instructions, such as multiply and divide instructions, require multiple cycles in the execute stage.

³ Branches that are not folded take an entry in the completion queue.

⁴ Some VFPU instructions skip VFPU-E1 and VFPU-E2, which may increase the latency of other VFPU instructions. See [Section 6.4.5.1.4, “Vector Floating-Point Unit \(VFPU\) Execution Timing.”](#)

Figure 6-4. e600 Core Pipeline Stages

6.3 Timing Considerations

When the fetch pipeline is full, as many as four instructions can be fetched to the IQ during each clock cycle. Three instructions can be dispatched to the issue queues per clock cycle.

The e600 core improves performance by executing multiple instructions at a time, using hardware to manage dependencies. When an instruction is issued, the register file or rename registers send the source

data to the appropriate reservation station. Register files and rename registers have sufficient bandwidth to allow dispatch of three instructions per clock cycle under most conditions.

The BPU decodes and executes branches immediately after they reach the IQ. When a branch cannot be resolved due to a CR, CTR, or LR dependency, the branch may be predicted and execution continues from the predicted path. If the prediction is incorrect, the following steps are taken:

1. The IQ is purged and fetching continues from the correct path.
2. If mispredicted instructions have entered the CQ, any instructions older than the predicted branch are allowed to retire, after which remaining instructions are purged from the CQ and execution units.
3. Dispatching resumes from the correct path.

After an instruction executes, results are made available to subsequent instructions in the appropriate rename registers. The architecture-defined GPR, FPR, and VR registers are updated during the write-back stage. Branch instructions that update the LR or CTR write back their results in a similar fashion.

After instruction execution, results are made available to subsequent instructions in the appropriate GPR, FPR, or VR rename registers. Results are then stored into the correct GPR, FPR, or VR during the write-back stage. If a subsequent instruction needs the result as a source operand, the result is simultaneously made available to the appropriate execution unit, which allows a data-dependent instruction to be decoded and dispatched without waiting to read the data from the register file. Branch instructions that update either the LR or CTR write back their results in a similar fashion.

The following section describes this process.

6.3.1 General Instruction Flow

As many as four instructions can be fetched into the IQ during each clock cycle. An instruction fetch consists of two single-cycle fetch stages—fetch1 and fetch2.

As many as three instructions can be dispatched per clock cycle, within the limitations of individual issue queues—the GIQ can accept as many as three, the FIQ can accept one, and the VIQ can accept as many as two instructions. Likewise, the GIQ can issue at most three instructions, the FIQ can issue one instruction, and the VIQ can issue two instructions per clock cycle. The e600 core tries to keep the IQ full at all times, unless instruction cache throttling is enabled, as described in [Section 9.7, “Instruction Cache Throttling.”](#)

The number of instructions fetched in a clock cycle is determined by the number of vacant spaces in the IQ during the previous clock cycle. This is shown in the examples in this chapter. If IQ8–IQ11 are available, the fetcher tries to initiate a fetch every cycle. Because the two fetch stages are pipelined, as many as four instructions can reach the IQ every cycle. However, the fetcher normally initiates a fetch only if IQ8–IQ11 are empty.

Typically, instructions are fetched from the L1 instruction cache, but as many as four instructions from a targeted stream can be fetched on a BTIC hit in 2 clock cycles, allowing the next four instructions to be fetched from the instruction cache with no idle cycles.

Branch instructions in IQ0–IQ7 are identified and forwarded to the BPU directly for immediate execution.

If a branch is predicted as taken, all instructions are flushed from the IQ in the next clock cycle. If the branch is unconditional or if the specified conditions are already known, the branch can be resolved immediately. That is, the branch direction and target address are known and instruction fetching can continue from the correct location. Otherwise, the branch direction or branch target address must be predicted. The e600 core offers several resources to resolve branch instructions and to improve the accuracy of branch predictions. These include the following:

- Branch target instruction cache (BTIC)—The 128-entry, four-way-associative BTIC, shown in [Figure 6-5](#), holds as many as four branch target instructions in each entry, so when a branch is encountered in a repeated loop, usually the first four instructions in the target stream can be fetched into the instruction queue on the next two clock cycles. The BTIC can be disabled and invalidated through bits in HID0.

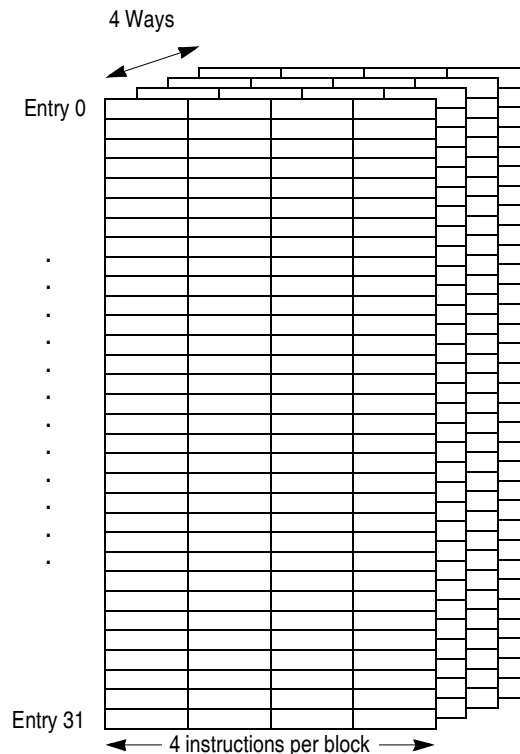


Figure 6-5. BTIC Organization

BTIC entries are indexed not from the address of the first target instruction but from the address of the branching instruction, so multiple branches sharing a target generate duplicate BTIC entries. Each entry can hold as many as four instructions, depending on where the first target instruction falls in the cache block.

As with other aspects of e600 core instruction timing, BTIC operation is optimized for cache-line alignment. If the first target instruction is one of the first five instructions in the cache block, the BTIC entry holds four instructions. If the first target instruction is the last instruction before the cache block boundary, it is the only instruction in the corresponding BTIC entry. If the next-to-last instruction in a cache block is the target, the BTIC entry holds two valid target instructions, as shown in [Figure 6-6](#).

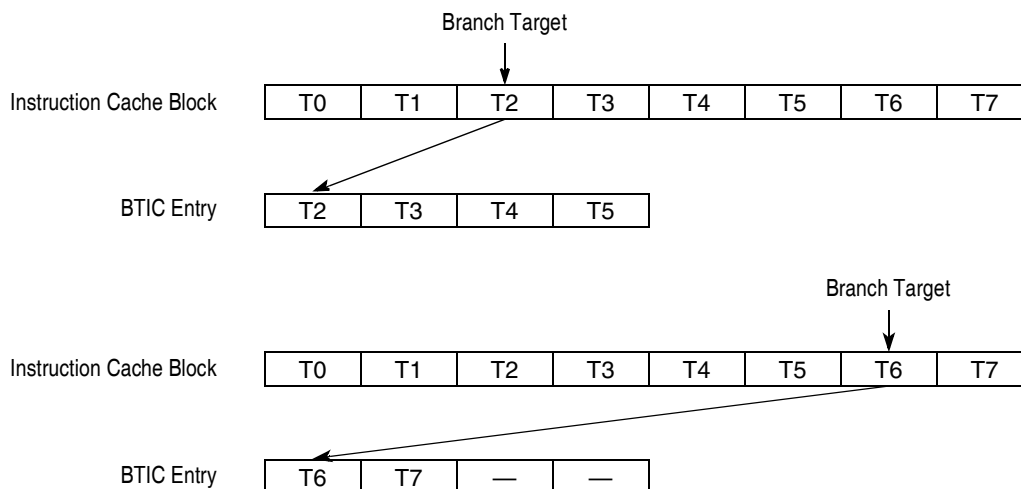


Figure 6-6. Alignment of Target Instructions in the BTIC

BTIC ways are updated using a FIFO algorithm.

Note that the entire BTIC is invalidated if translation changes (for example, a **tlbie** instruction executes, a TLB or BAT is updated, or an interrupt puts the processor in real mode) or if an **icbi** instruction invalidates an instruction cache block.

- Dynamic branch prediction—The 2048-entry branch history table (BHT) is implemented with 2 bits per entry for four levels of prediction—taken, not taken, strongly taken, and strongly not taken. Whether a branch instruction is taken or not taken can change the strength of the next prediction. Dynamic branch prediction is not defined by the PowerPC architecture.

To reduce aliasing, only predicted branches update BHT entries. Dynamic branch prediction is enabled by setting `HID0[BHT]`; otherwise, static branch prediction is used.

- Static branch prediction—Static branch prediction is defined by the PowerPC architecture and involves encoding the branch instructions. See 6.4.1.3.1, “Static Branch Prediction.”
- Link stack registers—The e600 core also avoid stalls by implementing an eight-entry branch link stack. As many as eight levels of **bclr**/branch-and-link pairs can be held and the **bclr** target address can be predicted from the link stack rather than requiring a stall until the **ld/mtlr** subroutine restore sequence completes.

The link register and rules required for correct use are as described in the *Programming Environments Manual*.

Correct use of the link stack requires that computed GOTO statements consist of the **mtctr/bcctr** instruction pair rather than the **mtlr/bclr** pair, which lowers link stack performance by requiring an expensive mispredict drain/flush as well as clearing the link stack to its initial empty state.

Attempting link stack and conditional branch prediction on the same instruction can affect performance. It may be necessary to avoid conditional **bclr** instructions because the BPU stalls execution until either the directional condition or the target address (LR) is resolved. Additionally, even if both are resolved when the conditional **bclr** is presented to the BPU, the conditional **bclr** takes 2 cycles to execute. Thus, except for code size, following a conditional branch (**bc**) with an unconditional **bclr** may be preferable to a conditional **bclr**.

Branch instructions that do not update the LR or CTR can be removed from the instruction stream, as described in [Section 6.4.1.1, “Branch Folding and Removal of Fall-Through Branch Instructions.”](#) Branch instructions that update the LR or CTR are treated as if they require dispatch, even though they are not dispatched to one of the issue queues. They must be assigned CQ entries to ensure that the CTR and LR are updated sequentially. The dispatch rate is affected by the serializing behavior of some instructions and on the availability of issue queues, execution units, rename registers, and CQ entries. Instructions are dispatched in program order; an instruction in IQ1 cannot be dispatched ahead of one in IQ0.

[Figure 6-7](#) shows instruction paths.

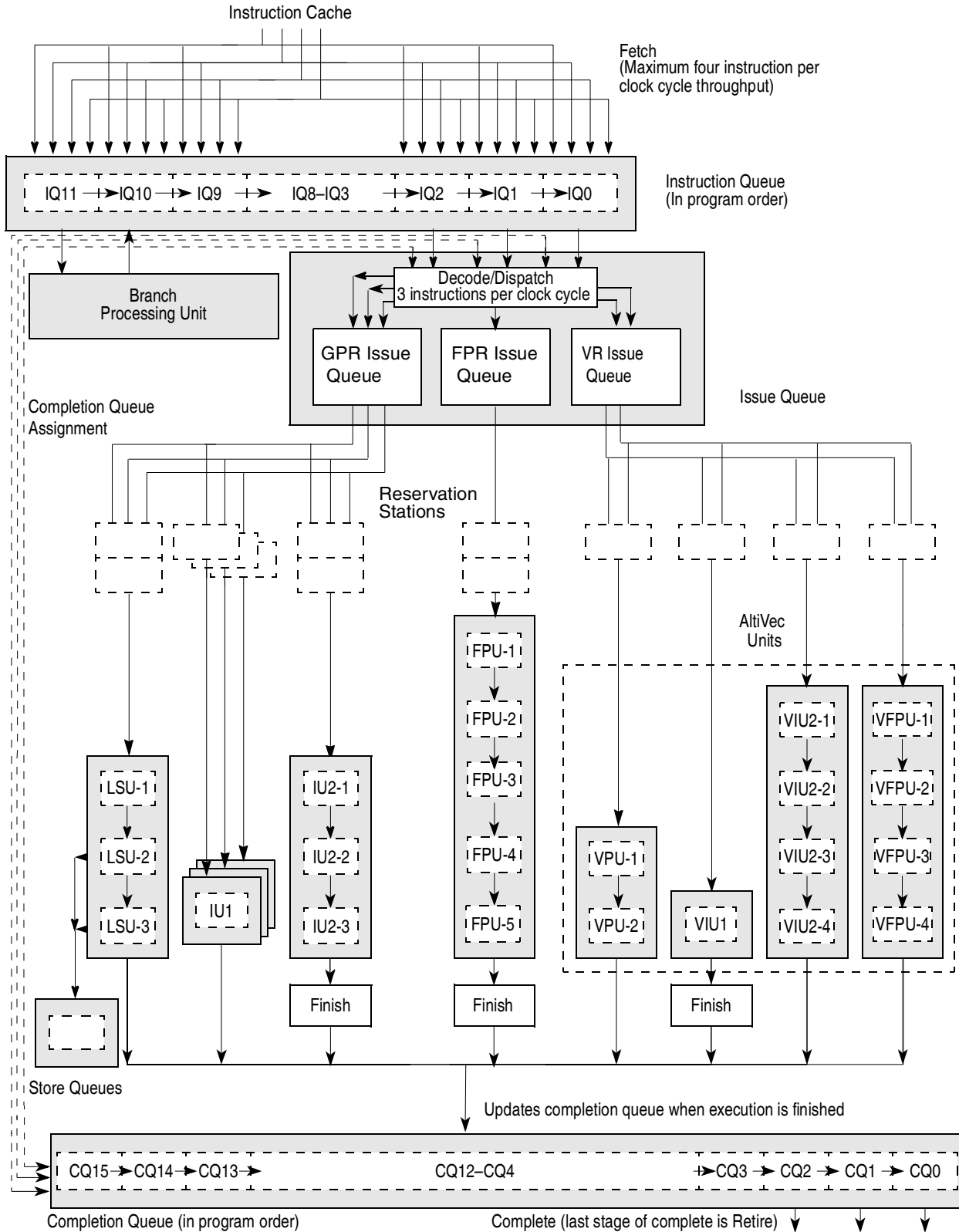


Figure 6-7. Instruction Flow Diagram

6.3.2 Instruction Fetch Timing

Instruction fetch latency depends on whether the fetch hits the BTIC, the instruction MMU, the L1 instruction cache, or the L2-cache within the core. If no cache hit occurs, a memory transaction is required in which fetch latency is affected by bus traffic and bus clock speed. These issues are discussed further in the following sections.

6.3.2.1 Cache Arbitration

When the instruction fetcher requests instructions from the instruction cache, two things may happen. If the access hits the instruction cache and the cache is idle, the instructions arrive two clock cycles later. However, if the cache or MMU is busy due to a higher priority operation, such as a **tlbie**, **icbi**, or a cache line reload, instructions cannot be fetched until that operation completes.

6.3.2.2 Cache Hit

If the instruction fetch hits the instruction cache, it takes only two clock cycles after the request for as many as four instructions to enter the IQ. Note that the cache is not blocked to internal accesses during a cache reload (hits under misses). The critical quad word is written simultaneously to the cache and forwarded to the requesting unit, minimizing stalls due to load delays. Note that the cache allows a hit under one miss and is only blocked by a cache line reload for the cycle when the cache write happens. So, if a cache miss is discarded by a misprediction and a new fetch hits, the cache allows instructions to come back. As many as four instructions are pipelined from the cache per cycle until the original miss comes back and a cache reload is performed, which blocks the cache for 1 cycle.

Figure 6-8 shows a simple example of instruction fetching that hits in the cache. This example uses a series of integer add instructions and double-precision, floating-point add instructions to show the following:

- How the number of instructions to be fetched is determined
- How program order is maintained by the IQ and CQ
- How instructions are dispatched, issued and completed
- How the FPU and IU2 pipelines function

Note that for the following instruction sequence instruction 0 is assumed to be the first instruction in the cache block. Also, there are no critical dependencies between instructions.

```

0      add
1      fadd
2      add
3      fadd
4      b 8
5      fsub
6      fadd
7      fadd
8      add
9      add
10     add
11     add
12     fadd
13     add
14     fadd
15     .
16     .
17     .
    
```

Instruction Timing

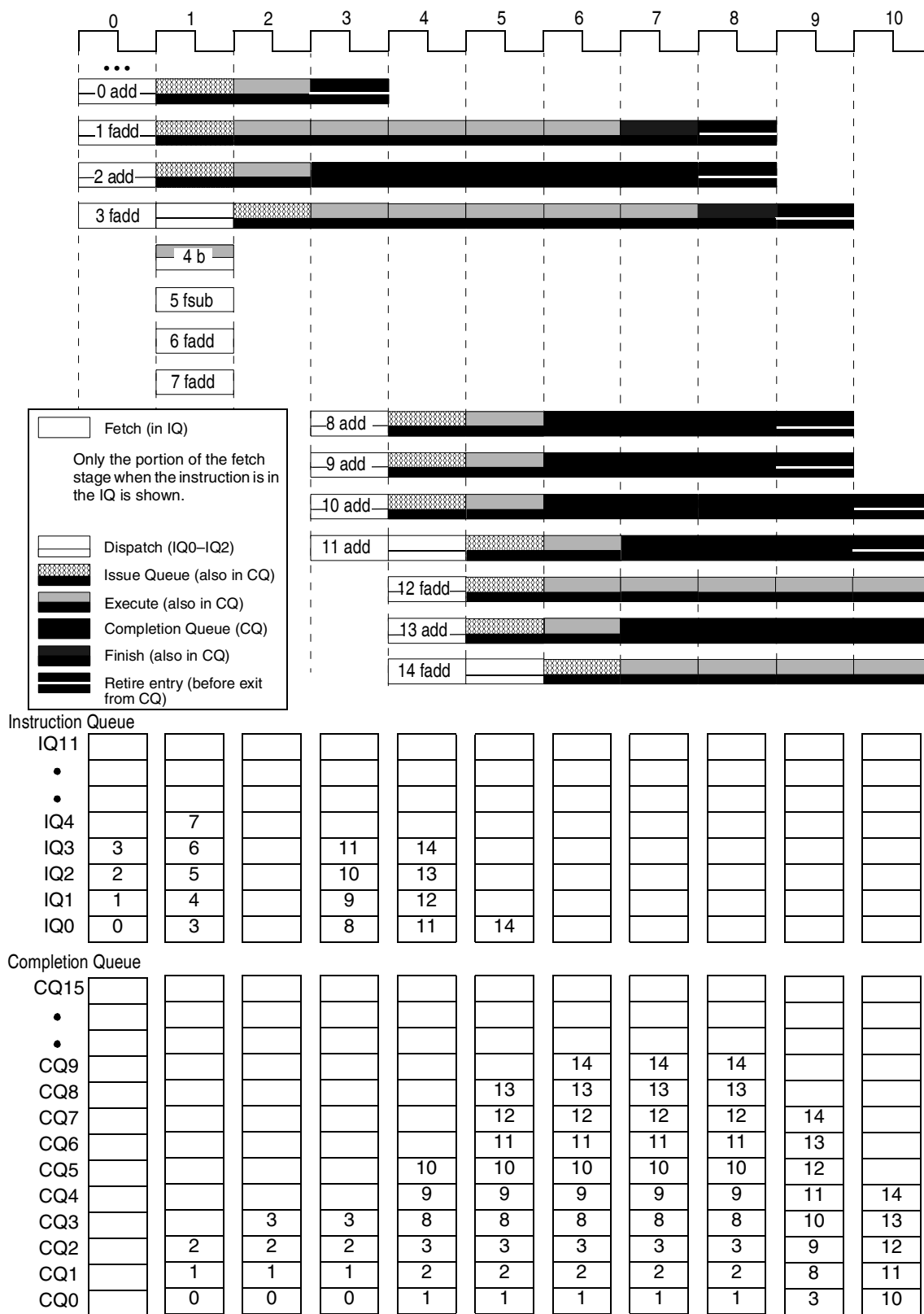


Figure 6-8. Instruction Timing—Cache Hit

The instruction timing for this example is described cycle-by-cycle as follows:

0. In cycle 0, instructions 0–3 are fetched from the instruction cache and are available in the IQ. These instructions are placed in IQ0–IQ3. Instructions 0–2 are dispatched at the end of this cycle to the GIQ, FIQ, and GIQ, respectively. As they are dispatched, they are also allocated in the bottom three entries of the CQ.
1. Instructions 0 and 2 are issued to two of the IU1s and instruction 1 is issued to the FPU. Note that because these instructions were dispatched in the last cycle, they have been assigned sequential positions in the CQ. In this case, instructions 0–2 are in the bottom three entries of the CQ. Instruction 3 is in IQ0 from which it is dispatched to the FIQ at the end of this cycle. Instructions 4–7 have arrived from the instruction cache at the end of the last cycle, which are also allocated in the IQ1–IQ4. The eight-instruction cache block boundary falls between instructions 7 and 8. Instruction 4 (the unconditional branch) is executed in this cycle and is immediately resolved as taken and therefore can be folded from the IQ.
2. Instructions 0 and 2 execute in the single-stage IU1s. Instruction 1 proceeds to the first of the five FPU stages. Instruction 3 is in the fourth entry of the CQ and is issued from the FIQ to the FPU. Because instruction 4 was an unconditional taken branch, it is folded from the IQ and instructions 5–7 are discarded. No new instructions are available because of the latency of the BTIC and instruction cache fetching.
3. Instruction 0 is retired at the end of this cycle. Instruction 1 proceeds to the second FPU stage. Instruction 2 has finished executing but must remain in the CQ until instruction 1 retires. Instruction 3 replaces instruction 1 in the first FPU stage. Instructions 8–11 are arrived from the BTIC and are also allocated in the bottom four entries of the IQ. Instructions 8–10 are dispatched to the GIQ at the end of this cycle.
4. Instruction 0 is retired from the CQ, so the remaining instructions are shifted down in the CQ. Instructions 1 and 3 are in the third and second FPU stages, respectively. Instruction 2 is still waiting in the CQ for instruction 1 to retire. Instructions 8–10 take positions in CQ. Instructions 12–14, which arrived from the instruction cache at the end of last cycle, are now in the IQ1–IQ3. Instructions 11–13 are dispatched at the end of this cycle.
5. Instructions 1 and 3 occupy to the fourth and third FPU stages, respectively. Instruction 2 waits in the CQ for instruction 1 to retire. Instructions 8–10 execute in the IU1s. Instructions 11–13 are issued and occupy CQ6–CQ8. Instruction 14 is dispatched at the end of this cycle.
6. Instruction 1 reaches the last FPU stage, while instruction 3 advances through the FPU pipeline. Instructions 2, 8, 9, and 10 wait in the CQ. Instructions 11–13 begin executing. Instruction 14 is issued to the FPU and occupies CQ9.
7. Instruction 1 occupies the FPU finish stage, while instruction 3 moves to the last FPU stage. Instructions 2, 8–11, and 13 wait in the CQ. Instruction 12 advances through the FPU pipeline, while instruction 14 moves to the first FPU stage.
8. Instructions 1 and 2 are retired at the end of this cycle. Instruction 3 moves to the FPU finish stage. Instructions 8–11 and 13 wait in the CQ. Instructions 12 and 14 advance through the FPU.
9. Because instructions 1 and 2 were retired in the last clock cycle, the remaining instructions shift down in the CQ and instructions 3, 8, and 9 are retired at the end of this clock cycle. Instructions 10, 11, and 13 wait in the CQ. Instructions 12 and 14 continue through the FPU.

10. Instructions 3, 8, and 9 retired in the last clock cycle, so the remaining instructions shift down in the CQ. Instructions 10 and 11 are retired at the end of this clock cycle, while instruction 13 waits in the CQ. Instructions 12 and 14 continue through the FPU.

6.3.2.3 Cache Miss

Figure 6-9 uses a similar instruction sequence as in Section 6.3.2.2, “Cache Hit,” but here the fetch misses the caches and requires an MPX bus access. Note that because the target instruction is not in the L1 cache, it cannot be in the BTIC. A 5:1 processor:bus clock ratio is used. The difference in the following code sample from that shown in Figure 6-8 is that the branch instruction 4 jumps to instruction 6 instead of instruction 8, and also the cache block boundary lies between instructions 5 and 6. The following example assumes the minimum possible cycles for an MPX bus transaction to occur, but any real system is likely to see further delays due to real DRAM latencies.

```

0      add
1      fadd
2      add
3      fadd
4      b 6
5      fsub
6      fadd
7      fadd
8      add
9      add
10     add
11     add
12     fadd
13     add
14     fadd
15     .
16     .
17     .

```

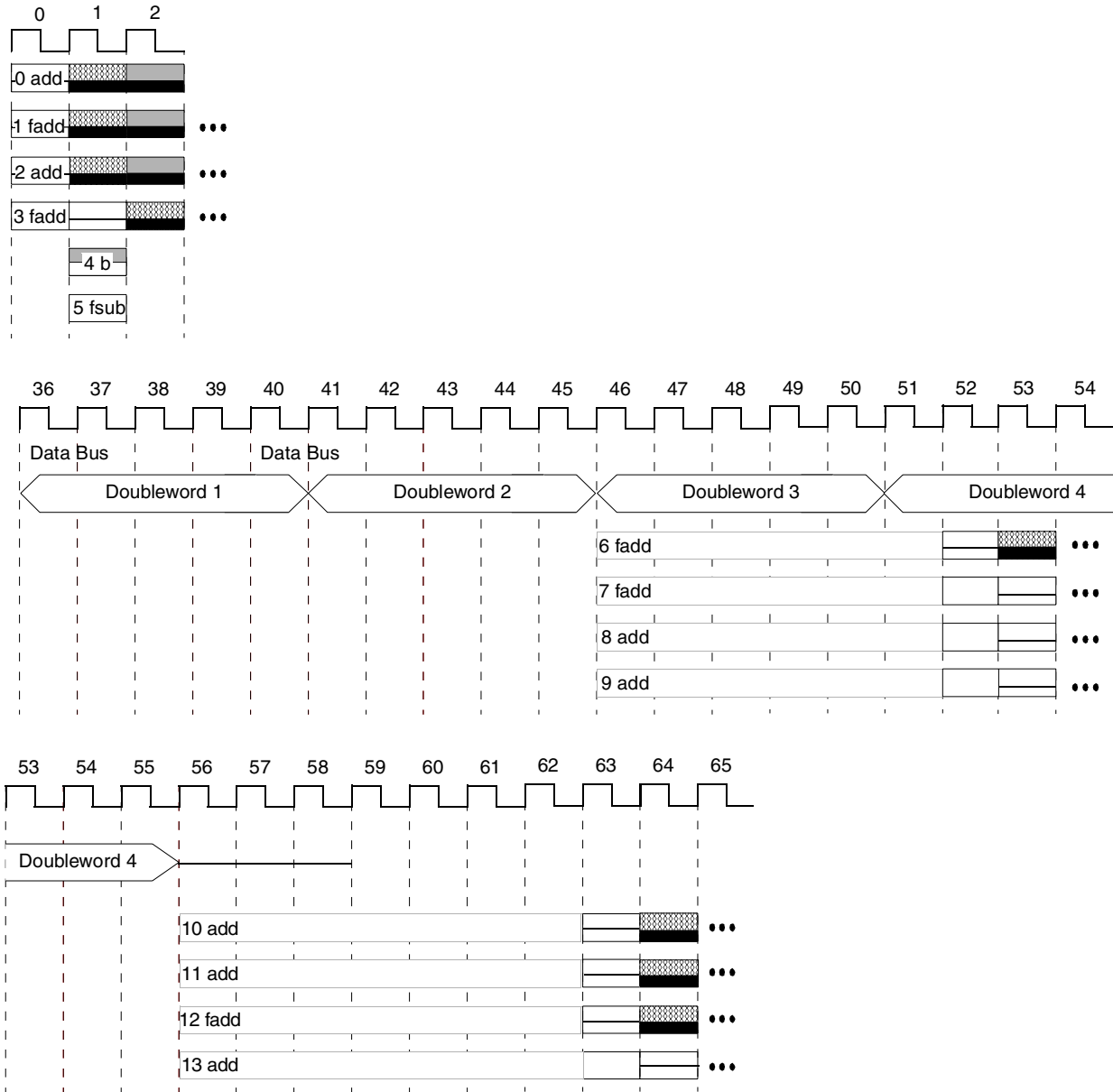


Figure 6-9. Instruction Timing—Cache Miss

A cache miss extends the latency of the fetch stage, so the fetch stage represents the time the instruction spends in the IQ and the time required for the instruction to be loaded from system memory, beginning in clock cycle 2.

The first four instructions follow the same pattern as in the cache hit example. The cache miss occurs in clock cycle 3, so the fetcher initiates a four-beat burst transaction to system memory. The critical quad word, which contains instructions 6–9, arrives in the first two beats and is forwarded to the IQ after the second beat. In clock cycle 52, these instructions are forwarded both to the instruction cache and to the

instruction fetcher. Instructions 10–13 arrive 11 cycles later in clock cycle 63. Assuming that instructions 14–17 are also not in any of the caches, they are available to the IQ in cycle 102. The second half of this cache block (instructions 18–21) arrives 11 cycles later (cycle 113).

6.3.2.4 L2 Cache Access Timing Considerations

If an instruction fetch misses the BTIC and the instruction cache within the core, the e600 core next looks in the L2 cache. If the requested instructions are there, they are burst into the e600 core in much the same way as shown in [Figure 6-9](#).

The example shown is for the fastest possible L2 response. Other factors can affect this latency, including whether the L2 cache is busy with other operations, like servicing a load request from the LSU.

6.3.2.4.1 Instruction Cache Miss/L2 Cache Hit

Full fetch latency (from fetch1 to arrival into IQ) for an instruction cache miss/L2 cache hit is 13 cycles. Note that an L2 hit provides a full cache line, so the instructions are fetched into the IQ back to back (see cycles 15 and 16 in [Figure 6-10](#)). [Figure 6-10](#) shows the same code sequence as [Figure 6-9](#).

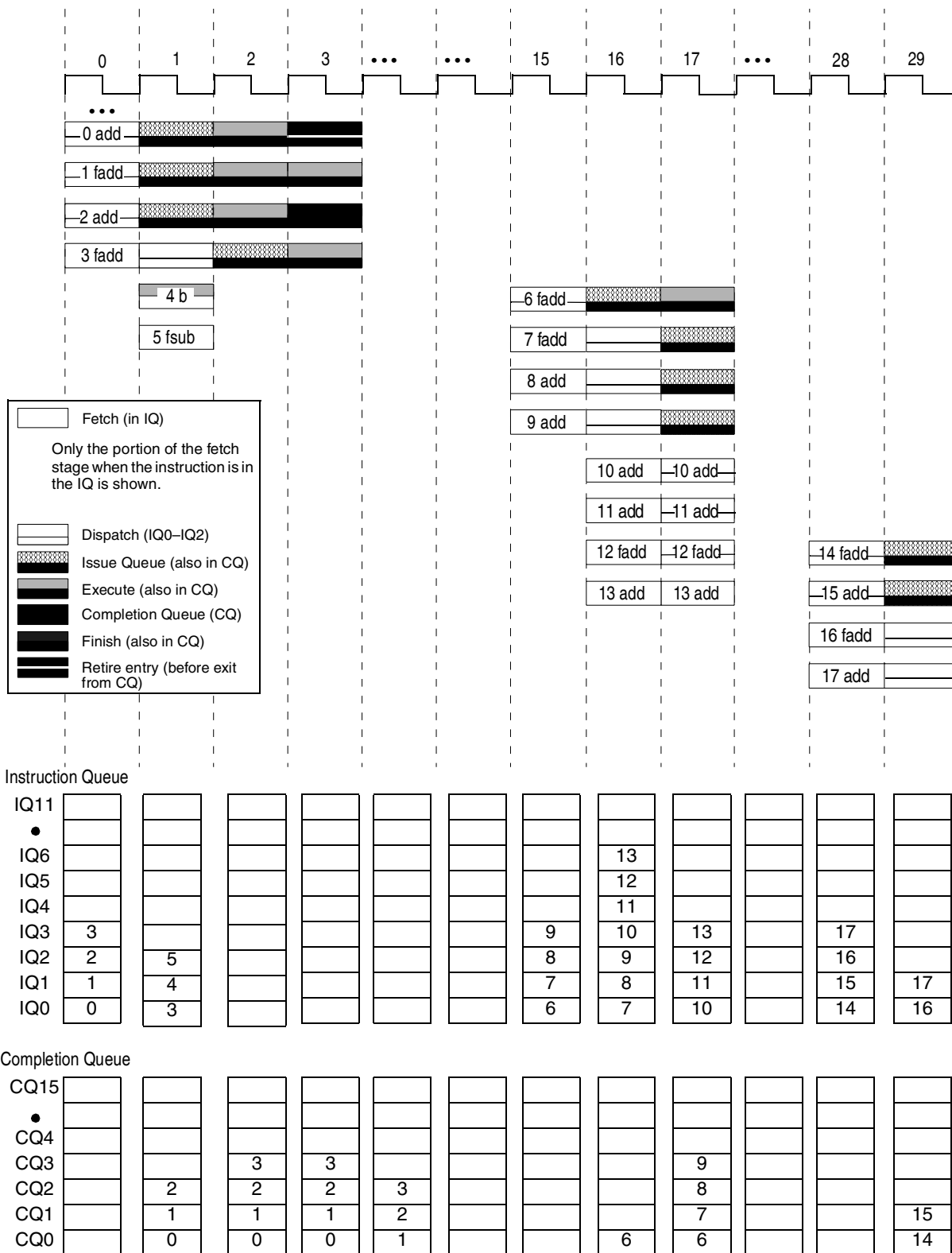


Figure 6-10. Instruction Timing—Instruction Cache Miss/L2 Cache Hit

6.3.3 Dispatch, Issue, and Completion Considerations

Several factors affect the core's ability to dispatch instructions at a peak rate of three per cycle—the mix of instructions and the availability of issue queues, destination rename registers, and CQ entries. Several of these factors are shown in the previous instruction timing examples.

Although as many as three instructions can be dispatched in parallel, they cannot be dispatched out of order; for example, an instruction in IQ1 cannot be dispatched unless the instruction in IQ0 can also dispatch.

To reduce issue unit stalls due to data dependencies, the LSU, IU2, and FPU have two-entry reservation stations; the other execution units have single-entry reservation stations. If a data dependency keeps an instruction from starting execution, that instruction is issued to a reservation station and the rename registers are assigned, eliminating the issue queue stalls. Execution begins during the same clock cycle that the rename buffer is updated with the data on which the instruction depends.

The issue queues allow the e600 core to dispatch decoded instructions even if execution units are busy. The dispatcher also allocates rename registers, locates source operands, and assigns rename registers for destination operands. The issue logic reads operands from register files and rename registers, and supplies rename tags for unavailable operands. The issue stage routes instructions to the proper execution unit. Execution begins when all operands are available, the instruction is in the bottom reservation station, and any execution serialization requirements are met.

The CQ maintains program order after instructions are dispatched, guaranteeing in-order completion and a precise interrupt model. In-order completion ensures the correct architectural state when the e600 core must recover from a mispredicted branch or interrupt.

Instruction state and other information required for completion are kept in the 16-entry, FIFO completion queue. Instructions cannot be retired ahead of previous instructions, as shown in [Section 6.3.2.2, “Cache Hit,”](#) and [Section 6.3.2.3, “Cache Miss.”](#)

Instructions are retired much as they are dispatched. As many as three instructions can be retired simultaneously, but instructions cannot be retired out of order. Note the following:

- Instructions must be non-speculative to complete.
- As many as three rename registers can be updated in a given register file. For example, a sequence of three **lfd** instructions, which requires three GPR rename registers and three FPR rename registers, can complete in one cycle. However, the sequence **lwzu, add, add** requires four GPR rename registers, so only the first two instructions can complete in a cycle.

Program-related interrupts are signaled when the instruction causing the interrupt reaches CQ0. Previous instructions are allowed to complete before the interrupt is taken, which ensures that any interrupts those instructions may cause are taken.

6.3.3.1 Rename Register Operation

To avoid contention for a given register file location in the course of out-of-order execution, the e600 core provides rename registers for holding instruction results before the completion commits them to the architecture-defined register. The GPRs, FPRs, and VRs each have 16 rename registers. The CR, LR, and CTR each have 1 rename register.

When an instruction is dispatched, a rename register (or registers) is allocated for the results of that instruction. If an instruction is issued to a reservation station because of a data dependency, issue logic also provides a tag to the execution unit identifying the rename register that forwards the required data at completion. Execution can begin when source data reaches the rename register.

Results from rename registers are transferred to the architecture-defined registers in the write-back stage. Renames are also deallocated in the write-back stage.

If branch prediction is incorrect, instructions after the branch are flushed from the CQ and any results of those instructions are flushed from the rename registers.

6.3.3.2 Instruction Serialization

Although the e600 core can dispatch and complete three instructions per cycle, serializing instructions limit dispatch and completion to one instruction per cycle. There are three basic types of instruction serialization:

- Execution serialization—Execution-serialized instructions are issued and held in the functional unit's reservation station. They do not execute until all prior instructions have completed. A functional unit holding an execution-serialized instruction does not accept further instructions from the issue queues. For example, execution serialization is used for instructions that modify non-renamed resources. Results from these instructions are generally not available or are forwarded to subsequent instructions until the instruction completes.
- Refetch serialization—Refetch-serialized instructions force refetching of subsequent instructions after completion. Refetch serialization is used when an instruction has changed or may change a particular context needed by subsequent instructions. Examples include **isync**, **sc**, **rfi**, **mtspr[XER]**, and any instruction that toggles the summary-overflow (SO) bit.
- Store serialization—(Applicable to stores and some LSU instructions that access the data cache.) Store-serialized instructions are dispatched and held in the LSU's finished store queue. They are not committed to memory until all prior instructions have completed. While a store-serialized instruction waits in the finished store queue, other load/store instructions can be freely executed. Store-serialized instructions complete only from CQ0, so only one store-serialized instruction can complete per cycle, although non-serialized instructions can complete in the same cycle as a store-serialized instruction. In general, all stores and cache operation instructions are store-serialized.

6.4 Execution Unit Timings

The following sections describe instruction timing considerations within each of the respective execution units in the e600 core.

6.4.1 Branch Processing Unit Execution Timing

Flow control operations (conditional branches, unconditional branches, and traps) are sometimes expensive to execute in most machines because they may disrupt normal instruction flow. When program flow changes, the IQ must be reloaded with the target instruction stream. Previously issued instructions continue executing while the new instruction stream makes its way into the IQ, but depending on whether

the target instruction is in the BTIC, instruction cache, L2 cache, or system memory, opportunities to execute instructions may be missed, as shown in the examples in [Section 6.3.2.3, “Cache Miss.”](#)

The penalties associated with e600 flow control operations are minimized by performance features such as branch folding, removal of fall-through branch instructions, BTIC, dynamic branch prediction (implemented in the BHT), three-level branch prediction, an eight-entry link stack, and the implementation of non-blocking caches. Timing for branch instruction execution is affected by whether the following occur:

- The branch is taken.
- Instructions in the target stream, typically the first four instructions in the target stream, are in the BTIC.
- The target instruction stream is in the cache within the core.
- The branch is predicted.
- The prediction is correct.

6.4.1.1 Branch Folding and Removal of Fall-Through Branch Instructions

After a branch enters the IQ, the BPU immediately begins to decode it and tries to resolve it. Except those that update the LR or CTR, most branch instructions are removed from the instruction flow before they take a position in the CQ.

Branch folding occurs either when a branch is predicted taken or is resolved taken (as in the case with unconditional branches). The cycle after branch execution, the BPU folds the branch out of the instruction stream (removes the branch from the IQ) and also flushes instructions in the IQ after the branch.

[Figure 6-11](#) shows branch folding with a BTIC hit and with a BTIC miss/instruction cache hit. Here a branch instruction **b** encountered in a series of **add** instructions is resolved as taken. What happens on the next clock cycle depends on whether the target instruction stream is cached or if an MPX bus transfer is required.

If there is a BTIC hit, the **b** instruction is folded on the next clock cycle and the instructions after the branch (**add4** and **add5**) are flushed. On the cycle after that (clock2), instructions **xor1–xor4** have arrived from the BTIC, and **xor1–xor3** are dispatched. In clock 3, **xor5–xor8** arrive.

If the target instructions are not in the BTIC, it takes 2 cycles (fetch1 and fetch2) to attempt to fetch the first four instructions from the instruction cache. These instructions arrive and are ready for dispatch in clock 3.

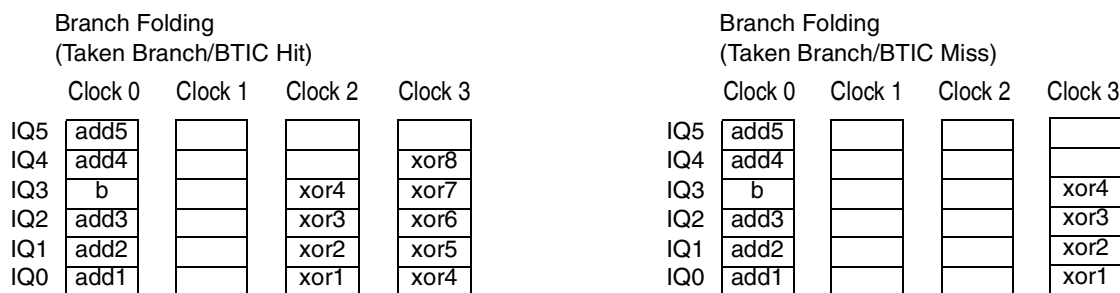


Figure 6-11. Branch Folding

The effect of the taken branch on the instruction supply is known as the branch-taken bubble. For the BTIC hit case, there is a 1-cycle branch-taken bubble (clock1). For the BTIC miss case, there is a 2-cycle bubble (clocks 1 and 2).

If the fetch misses all of the caches, a system memory access is required, the latency of which depends on factors such as processor:bus clock ratio. In most cases, a memory access indicates that execution units remain idle, as shown in [Section 6.3.2.3, “Cache Miss.”](#)

[Figure 6-12](#) shows the attempted removal of fall-through branch instructions, which can occur when a branch is not taken or is predicted as not taken.

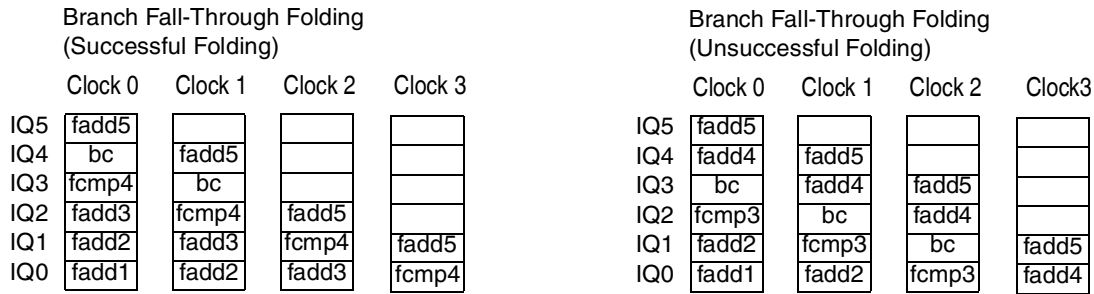


Figure 6-12. Removal of Fall-Through Branch Instruction

A not-taken branch instruction stays in the IQ for at least the cycle after execution. If it does not update the LR or CTR and is in IQ3–IQ7 in the cycle after execution, it can be removed from the IQ. In [Figure 6-12](#), the branch is predicted as not taken and executes in cycle 0 in both cases. When the branch is in IQ3, as in the successful case on the left, it is removed from the IQ (clock 2). In the unsuccessful case on the right, it reaches IQ2 in clock 1, from which it cannot be removed, and so must be dispatched.

When a correctly predicted taken branch instruction is detected before reaching a dispatch position, folding the branch instruction and flushing any instructions from the incorrect path may eliminate any latency required for control flow. Execution proceeds as though the branch were never there. However, in many cases the branch-taken bubble may waste a few dispatch opportunities.

The advantage of removing not-taken branch instructions (fall-through) is slightly less than that of branch folding. Although the branch may be removed from the instruction stream and not require a dispatch slot, if the branch reaches one of the dispatch entries (IQ0–IQ2) it requires both a dispatch slot and a CQ entry.

6.4.1.2 Branch Instructions and Completion

As described in the previous section, instructions that do not update the LR or CTR are often removed from the instruction stream before they reach the CQ. However, branch instructions that update the architecture-defined LR and CTR must write back in program order like the instructions that update the FPRs, GPRs, and VRs.

Executed branch instructions that are not removed are not sent to an issue queue at dispatch but are assigned a CQ slot, as shown in [Figure 6-13](#).

	Clock 0	Clock 1	Clock 2	Clock 3	Clock 4	Clock 5
IQ5	add4	add5				
IQ4	add3	add4				
IQ3	add2	add3				
IQ2	add1	add2	add5			
IQ1	bc	add1	add4			
IQ0	cmp	bc	add3			
CQ7						
CQ6				add5		
CQ5				add4		
CQ4				add3	add5	
CQ3			add2	add2	add4	
CQ2			add1	add1	add3	
CQ1			bc	bc	add2	add5
CQ0		cmp	cmp	cmp	add1	add4

Figure 6-13. Branch Completion (LR/CTR Write-Back)

In this example, the **bc** that executes in clock cycle 0 depends on **cmp** and is predicted as not taken. Because the branch executes in clock cycle 0, it cannot be dispatched. Because the branch is in IQ0 in the cycle after execution, it cannot be folded and is dispatched with **add1** and **add2**. The **cmp** executes in clock cycle 2 and the branch resolves as correctly predicted in clock cycle 3.

Additionally, **cmp** and **bc** retire at the end of clock cycle 3. Even if **add1** were finished, it could not retire because the **bc** is resolving in this cycle. In the cycle after branch resolution, cycle 4, **add1–add5** are marked as non-speculative and **add1–add3** are allowed to retire.

6.4.1.3 Branch Prediction and Resolution

The e600 core supports the following two types of branch direction prediction:

- Static branch prediction—This is defined by the PowerPC architecture as part of the encoding of branch instructions.
- Dynamic branch prediction—This is a processor-specific mechanism implemented in hardware (in particular the branch history table, or BHT) that monitors branch instruction behavior and maintains a record from which the next occurrence of the branch instruction is predicted.

When a conditional branch direction cannot be resolved due to a CR or CTR data dependency, the BPU predicts whether it will be taken and instruction fetching proceeds down the predicted path. If the prediction is wrong, subsequent instructions and their results are purged. Instructions ahead of the predicted branch proceed normally, and instruction fetching resumes along the correct path.

The e600 core executes through three prediction levels. Instructions from all three unresolved branches are allowed to execute but cannot complete until all older branches are resolved. If three predicted branches are outstanding, no further conditional branches can be processed (although BPU processes any unconditional branch). When one or more of the three previous conditional branches is resolved, the BPU can begin processing new conditional branches.

The number of instructions that can be executed after the issue of a predicted branch instruction is limited by the fact that no instruction executed after a predicted branch can update the register files or memory until the branch is resolved. That is, instructions may be issued and executed, but cannot reach the write-back stage. When an instruction in a predicted branch stream finishes, it does not write back its results to the architecture-defined registers until the branch is resolved, which may cause a stall in the CQ.

In case of a misprediction, the e600 core can easily redirect its machine state because the programming model has not been updated. If a branch is mispredicted, all instructions dispatched after the predicted branch instruction are flushed from the CQ and any results are flushed from the rename registers.

If the search for the branch target hits in the BTIC, one of the four target instructions is available in the IQ 2 cycles later (shown in [Figure 6-11](#)). The BTIC is described in detail in [Section 6.3.1, “General Instruction Flow.”](#)

In some situations, an instruction sequence creates dependencies that keep a branch instruction from being resolved immediately, thereby delaying execution of the subsequent instruction stream based on the prediction. The instruction sequences and the resulting action of the branch instruction are as follows:

- An **mtspr**(LR) followed by a **bclr**—The link stack is used to predict the instruction target address.
- An **mtspr**(LR) followed by a conditional **bclr**—The BPU stalls this branch until either the LR becomes available or the required CR/CTR data becomes available. The BPU can predict a branch either on a direction basis (CR/CTR) or an address basis (LR) but not both simultaneously. Note that a conditional **bclr** also requires 2 cycles to execute in the BPU.
- An **mtspr**(CTR) followed by a **bcctr**—Fetching stops and the branch waits for the **mtspr** to execute.
- A fourth conditional branch—A fourth conditional branch is encountered while three branches are unresolved (based on CR and CTR direction predictions or LR address predictions). The fourth **bc** is not executed and the BPU stalls. This normally forces the fetcher to stall a cycle or two later when the IQ fills up behind the stalled branch. This stall continues until at least one unresolved branch resolves. Note that branch conditions can be a function of the CTR and the CR; if the CTR condition is sufficient to resolve the branch, any CR dependency is ignored.

6.4.1.3.1 Static Branch Prediction

The PowerPC architecture provides a field in branch instructions (the BO field) to allow software to hint whether a branch is likely to be taken. Rather than delaying instruction processing until the condition is known, the e600 core begins fetching and executing along the predicted path. When the branch condition is known, the prediction is evaluated. If the prediction is correct, program flow continues along that path; otherwise, the processor flushes any instructions and their results from the mispredicted path, and program flow resumes along the correct path.

Static branch prediction is used when `HID0[BHT]` is cleared, which disables the branch history table. For information about static branch prediction, see the sections “Conditional Branch Control,” and “Addressing Modes and Instruction Set Summary,” in the *Programming Environments Manual*.

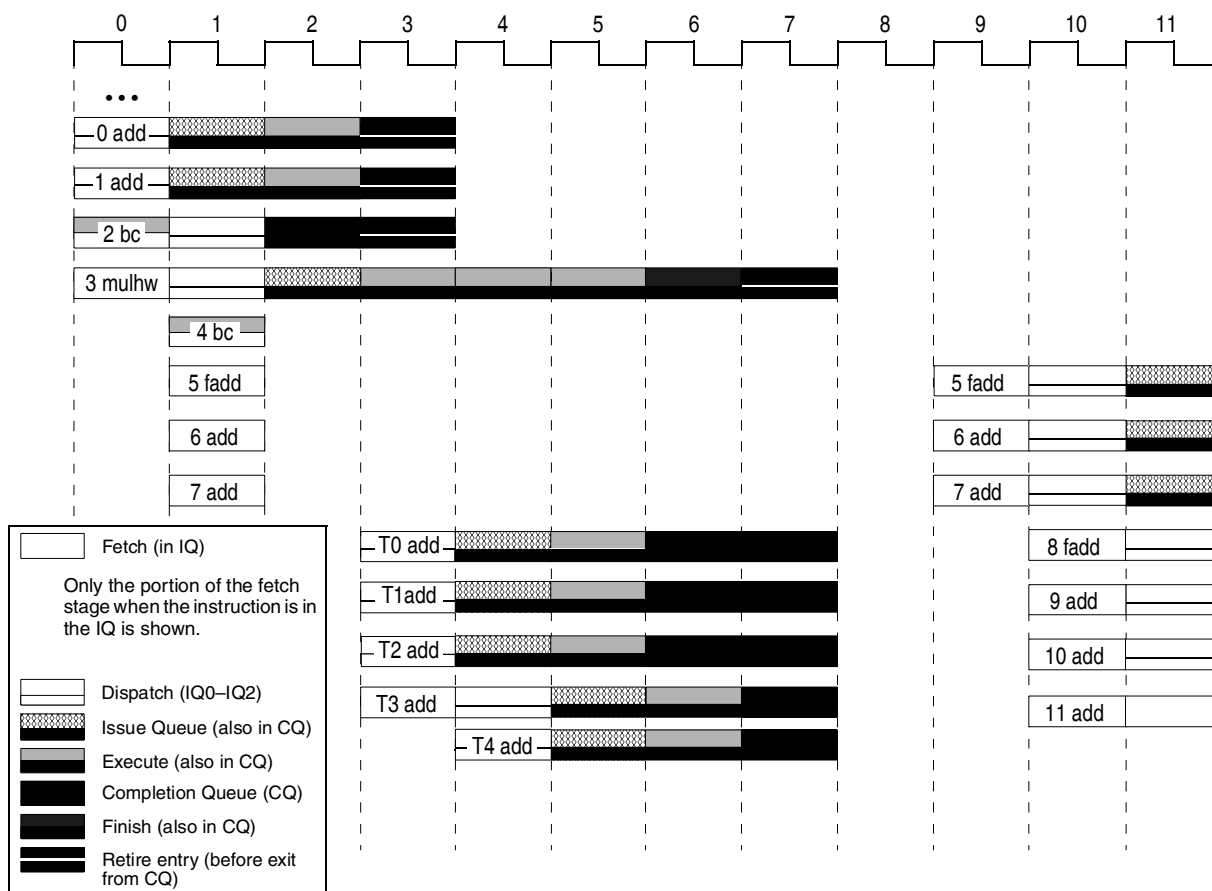
6.4.1.3.2 Predicted Branch Timing Examples

Figure 6-14 shows cases where branch instructions are predicted. It shows how both taken and not-taken branches are handled and how the e600 core handles both correct and incorrect predictions. The example shows the timing for the following instruction sequence:

```

0      add
1      add
2      bc
3      mulhw.
4      bc T0
5      fadd
6      add
7      add
T0     add
T1     add
T2     add
T3     add
T4     and
T5     add

```



Instruction Queue

IQ11											
•											
•											
IQ5		7									
IQ4		6									
IQ3	3	5		T3							11
IQ2	2	4		T2					7		10
IQ1	1	3		T1	T4				6		9
IQ0	0	2		T0	T3				5		8

Completion Queue

CQ15											
•											
•											
CQ5						T4	T4	T4			
CQ4						T3	T3	T3			
CQ3			3	3	T2	T2	T2	T2			
CQ2			2	2	T1	T1	T1	T1			7
CQ1		1	1	1	T0	T0	T0	T0			6
CQ0		0	0	0	3	3	3	3			5

* Instructions 5 and 6 are not in the IQ in clock cycle 5. Here, the fetch stage shows cache latency.

Figure 6-14. Branch Instruction Timing

Instruction timing for this example is described cycle-by-cycle as follows:

0. At the end of clock cycle 0, instructions 0 and 1 are dispatched. Instruction 2, a branch instruction that updates the CTR, is predicted as not taken. Instruction 3 is in IQ3.
1. Instructions 0 and 1 are in the GIQ. After the branch instruction is executed, it goes into the CQ from which it updates the CTR when it retires. At the end of this cycle, instruction 3 is dispatched to the GIQ. Because the second **bc** instruction (instruction 4) is predicted as taken, it can be folded. Because it is a BTIC hit, the target instruction stream can be fetched and available for dispatch at the end of clock cycle 3. Although the second **bc** is predicted as taken, it continues through the IQ until the misprediction is detected. Instructions 5–7 are fetched. Because the second branch (instruction 4) is predicted as taken, instructions 5–7 are removed from the IQ at the end of this cycle.
2. Instructions 0 and 1 enter two IU1s. Instruction 2 must remain in the CQ until instructions 0 and 1 complete. Instruction 3 is issued to the IU2.
3. Instructions 0–2 are retired. Instruction 3 continues through the IU2. The BTIC provides instructions T0–T3 to the IQ as branch instruction 4 is executed, predicted taken, and folded. T0–T2 are dispatched at the end of this cycle.
4. Instruction 3 recognizes an early-out condition and moves to the final IU2 stage. Instructions T0–T2 are issued to the three IU1s, and T3 and T4 are dispatched at the end of this clock cycle.
5. Instruction 3 moves to the final IU2 stage. T0–T2 are executed in the IU1s. Instructions T3 and T4 are issued at the end of this clock cycle.
6. Instruction 3 enters the IU2 finish stage. T3 and T4 execute in two IU1s, and T0–T2 wait in the CQ.
7. Instruction 3 is retired and its results indicate that the branch (instruction 4) was mispredicted, so instructions T0–T4 are flushed at the end of this cycle.
8. The mispredicted branch is resolved, so instructions 5–7 are refetched and are now in the fetch1 stage.
9. Instructions 5–7 are in fetch2 and instructions 8–11 are in fetch1.
10. Instructions 5–7 are dispatched at the end of this cycle and instructions 8–11 are in fetch2.
11. Instructions 5–7 are issued and instructions 8–11 are dispatched.

6.4.2 Integer Unit Execution Timing

The e600 core has three single-cycle integer units (IU1s) and one multiple-cycle integer unit (IU2). The three IU1s execute all integer instructions except multiplies and divides. The IU2 executes multiplies, divides, and several miscellaneous instructions, including CR logic and move to/from SPR instructions. [Table 6-5](#) lists integer unit instruction latencies. As [Table 6-5](#) shows, most integer instructions have a single-cycle execution latency.

6.4.3 FPU Execution Timing

The FPU executes single- and double-precision floating-point operations compliant with the IEEE-754 floating-point standard. Single- and double-precision floating-point multiply, add, and subtract execute in a five-stage pipeline. Most floating-point instructions execute with 5-cycle latency and 1-cycle throughput; however, **fdivs**, **fres**, and **fdiv** have latencies of 14 to 35 cycles. The **fdivs**, **fdiv**, **fres**, **mcrfs**,

mtfsb0, **mtfsb1**, **mtfsfi**, **mffs**, and **mtfsf** instructions block the FPU pipeline until they complete execution, inhibiting the issue of additional floating-point instructions. [Table 6-6](#) shows floating-point instruction execution timing.

6.4.3.1 Effect of Floating-Point Exceptions on Performance

For the best, most predictable e600 core floating-point performance, IEEE Std. 754 floating-point exceptions should be disabled in the FPSCR and MSR.

If an exception is enabled (through a combination of MSR[FE0, FE1] and one or more FPSCR enable bits), the instruction traps to the program interrupt handler. Floating-point operations that create an exception sticky bit in the FPSCR may degrade performance.

6.4.4 Load/Store Unit Execution Timing

The LSU executes load and store instructions, including AltiVec LRU and transient instructions. The execution of most load instructions is pipelined in the three LSU stages, during which the effective address is calculated, MMU translation is performed, the data cache array and tags are read, and cache way selection and data alignment are carried out. If there are no data dependencies, cacheable GPR and vector register load instructions have a 3-cycle latency and a 1-cycle throughput. Cacheable FPR load instructions have a 4-cycle latency and a 1-cycle throughput. The data cache array is updated after a store instruction is retired.

If operands are misaligned, additional latency may be required either for an alignment interrupt or for additional MPX bus accesses. Load instructions that miss in the cache block subsequent cache accesses during the cache line refill. [Table 6-7](#) gives load and store instruction execution latencies.

6.4.4.1 Effect of Operand Placement on Performance

The location and alignment of operands in memory may affect performance of memory accesses, in some cases significantly, as shown in [Table 6-1](#).

Alignment of memory operands on natural boundaries guarantees the best performance. For the best performance across the widest range of implementations, the programmer should assume the performance model described in Chapter 3, “Operand Conventions,” in the *Programming Environments Manual*.

The effect of alignment on memory operation performance is the same for big- and little-endian addressing modes except for multiple and string operations, which cause an alignment interrupt in little-endian mode.

In [Table 6-1](#), ‘optimal’ means that one effective address (EA) calculation occurs during the memory operation. ‘Fair’ means that multiple EA calculations occur during the operation, which may cause additional MPX bus activities with multiple MPX bus transfers. ‘Poor’ means that an alignment interrupt is generated by the memory operation.

Table 6-1. Performance Effects of Memory Operand Placement

Operand		Boundary Crossing ¹			
Size	Byte Alignment	None	8 Byte	Cache Line	Protection Boundary
Integer					
4 byte	4 <4	Optimal Optimal	— Fair	— Fair	— Fair
2 byte	2 <2	Optimal Optimal	— Fair	— Fair	— Fair
1 byte	1	Optimal	—	—	—
lmw, stmw ²	4 <4	Fair Poor	Fair Poor	Fair Poor	Fair Poor
String ^{2, 4}		Fair	Fair	Fair	Fair
Floating-Point					
8 byte	8 4 <4	Optimal — —	— Fair Poor	— Fair Poor	— Fair Poor
4 byte	4 <4	Optimal Poor	— Poor	— Poor	— Poor
AltiVec ³					
16 byte	16	Optimal	—	—	—

¹ Vector operands are not shown because they are always aligned.
 Optimal: One EA calculation occurs.
 Fair: Multiple EA calculations occur which may cause additional MPX bus activities with multiple MPX bus transfers.
 Poor: Alignment interrupt occurs.

² These operations are not supported in little-endian mode and will cause an alignment interrupt.

³ AltiVec memory operations are forced to align on a 16-byte boundary.

⁴ Usage of string instructions is strongly discouraged.

6.4.4.2 Store Gathering

The e600 core performs store gathering for cache-inhibited and write-through operations to nonguarded space as well as for cacheable write-back stores. However, stores are gathered only if the successive stores meet the criteria and are queued and pending. Store gathering occurs regardless of the address order of the stores and is enabled by setting HID0[SGE]. Bytes can be gathered into half words, which can be gathered into words, which can be gathered into double words, which can be gathered into quad words, and quad words can be gathered into cache lines. In addition, cacheable write-back stores to the same cache line can be merged regardless of size or alignment.

Store gathering is not done for the following:

- Stores to guarded cache-inhibited or write-through space
- **stwcx.** instructions
- Cache-inhibited or write-through stores if the result would violate MPX bus alignment

If store gathering is enabled and the stores do not fall under the above categories, an **eieio** or **sync** instruction must be used to prevent two stores from being gathered.

6.4.4.3 AltiVec Instructions Executed by the LSU

The LSU executes the AltiVec LRU and transient instructions.

6.4.4.3.1 LRU Instructions

The AltiVec architecture specifies that the **lvxl** and **stvxl** instructions differ from other AltiVec load and store instructions in that they leave cache entries in a least-recently-used state instead of a most-recently-used state. This is used to identify data that is known to have little reuse and poor caching characteristics.

On the e600 core, these instructions follow the cache allocation and replacement policies described in [Chapter 3, “L1 and L2 Cache Operation,”](#) but they leave their addressed cache entries in the LRU state. In addition, all LRU instructions are also interpreted to be transient and are also treated as described in the next section. Additional discussion of LRU effects can be found in [Chapter 3, “L1 and L2 Cache Operation.”](#)

6.4.4.3.2 Transient Instructions

The AltiVec architecture describes a difference between static and transient memory accesses. A static memory access should have some reasonable degree of locality and should be referenced several times or reused over some reasonably long period of time. A transient memory reference has poor locality and is likely to be referenced very few times or over a very short period of time.

The e600 core supports both static and transient memory access behavior. If a memory access is designated as transient, that cache block is not allocated into the L2. As L1 castouts are not consumed by the L2 cache unless the line is already resident in the L2 cache, this forces the block to be written directly to main memory, bypassing the L2 cache.

The following instructions are interpreted to be transient:

- **dstt** and **dststt** (transient forms of the two data stream touch instructions)
- **lvxl** and **stvxl**

Use of the **dststt** and **dststt** instructions is not recommended. A **dst** or **dstt**, or sometimes a series of **dcbz** instructions, is almost always more appropriate.

6.4.5 AltiVec Instructions

The e600 core implements all instructions in the AltiVec specification. The AltiVec instruction set has no optional instructions; however, a few instructions associated with the load/store model are defined to allow significant differences between implementations. The following sections describe the e600 core implementation of these options.

6.4.5.1 AltiVec Unit Execution Timing

The AltiVec unit contains the following four independent execution units:

- Vector permute unit (VPU)—All AltiVec permute instructions are executed in 2 cycles.
- Vector simple integer unit (VIU1)
- Vector complex integer unit (VIU2)
- Vector floating-point unit (VFPU)

Execution timing for these units is described in the following sections.

6.4.5.1.1 AltiVec Permute Unit (VPU) Execution Timing

The VPU executes all AltiVec permute instructions, which have a 2-cycle latency.

6.4.5.1.2 Vector Simple Integer Unit (VIU1) Execution Timing

The VIU1 executes all AltiVec simple integer instructions, all of which have a single-cycle latency plus a separate finish stage.

6.4.5.1.3 Vector Complex Integer Unit (VIU2) Execution Timing

The VIU2 executes all AltiVec complex integer instructions, which have a 4-cycle latency.

6.4.5.1.4 Vector Floating-Point Unit (VFPU) Execution Timing

Most AltiVec floating-point instructions (regardless of Java/non-Java mode) have a 4-cycle latency on the e600 core. However, the **vcmpbfp**, **vcmpeqfp**, **vcmpgtfp**, **vmaxfp**, and **vminfp** instructions have 2-cycle latency. Under the conditions shown in [Figure 6-16](#), these instructions may increase the latency of other VFPU instructions.

The following two examples show a VFPU pipelining special case for vector-float-compare instructions.

In the following code sequence, **vcmpbfp** takes only 2 cycles. Note that the **vcmpbfp** jumps from execution stage 0 in clock cycle 2 to execution stage 3 in clock cycle 3. However, the bypass does not block other instructions.

```
0 vcmpbfp
1 vaddfp
2 vaddfp
```

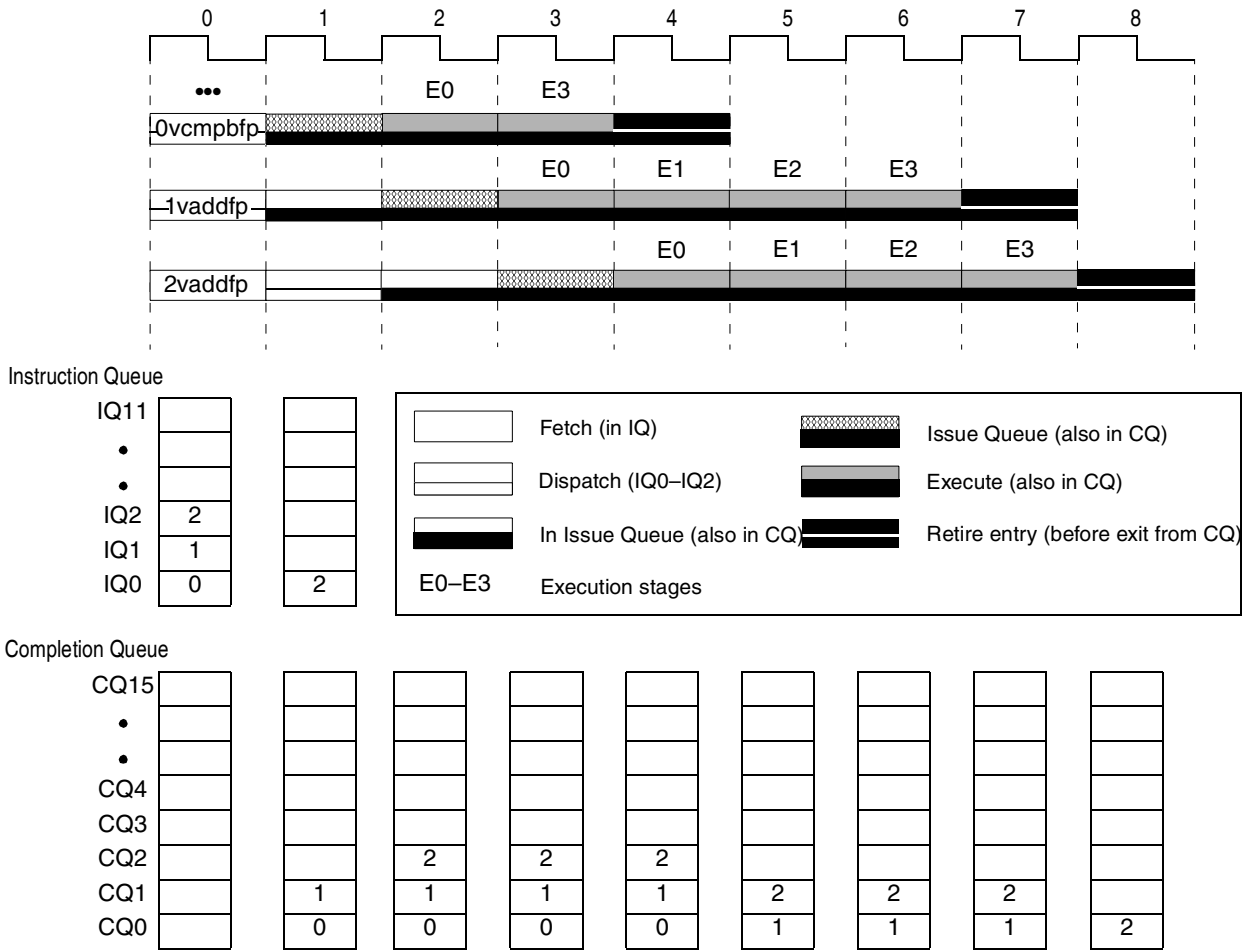
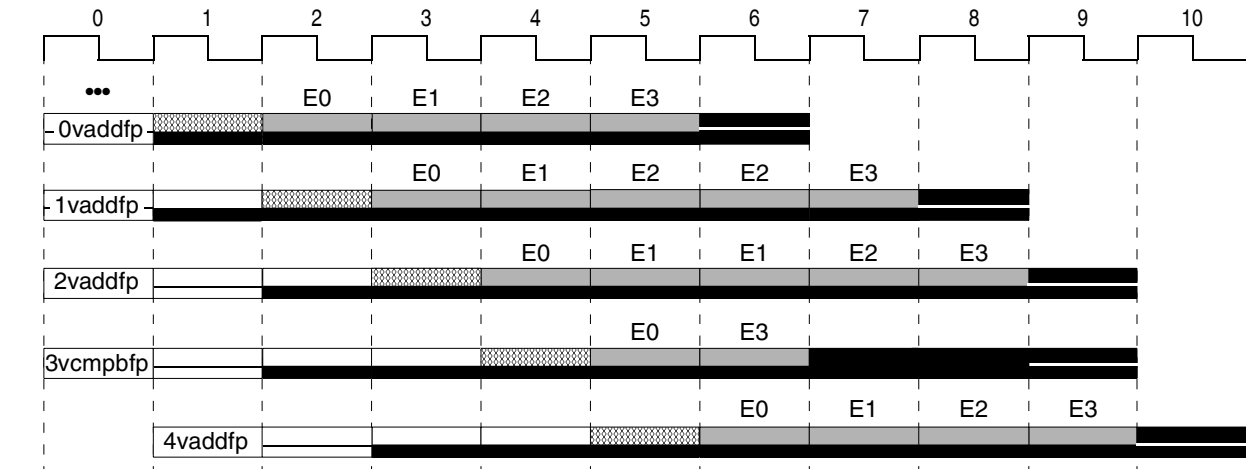


Figure 6-15. Vector Floating-Point Compare Bypass Non-Blocking

In the next sequence, **vcmpbfp** has a 2-cycle latency and bypasses other instructions. Note that in clock cycle 5, the bypass blocks all other instructions in the sequence. Because of the bypass, the second and third **vaddfp** do not advance in that cycle. Effectively, the bypass causes the second and third **vaddfp** to have a 5th cycle of latency.

- 0 vaddfp
- 1 vaddfp
- 2 vaddfp
- 3 vcmpbfp
- 4 vaddfp



Instruction Queue

IQ11			
•			
•			
IQ3	3		
IQ2	2	4	
IQ1	1	3	
IQ0	0	2	4

	Fetch (in IQ)		Execute (also in CQ)
	Dispatch (IQ0–IQ2)		Completion Queue (CQ)
	In Issue Queue (also in CQ)		Retire entry (before exit from CQ)
	Issue Queue (also in CQ)	E0–E3	Execution stages

Completion Queue

CQ15										
•										
•										
CQ4			4	4	4	4				
CQ3			3	3	3	3	4	4		
CQ2			2	2	2	2	3	3	4	
CQ1		1	1	1	1	1	2	2	3	
CQ0		0	0	0	0	0	1	1	2	4

Figure 6-16. Vector Float Compare Bypass Blocking

6.5 Memory Performance Considerations

Because the e600 core has a maximum instruction throughput of three instructions per clock cycle, lack of memory bandwidth can affect performance. To maximize performance, the e600 core must be able to read and write data efficiently. If a system has multiple bus devices, one device may experience long memory latencies while another device (for example, a direct-memory access controller) is using the MPX bus.

6.5.1 Caching and Memory Coherency

To minimize the effect of bus contention, the PowerPC architecture defines WIMG bits that define caching characteristics for the corresponding page or block. Accesses to caching-inhibited memory locations never update the L1 or L2 caches. If a cache-inhibited access hits in any of the caches, the cache block is

invalidated. If the cache block is marked modified, it is copied back to memory before being invalidated. Where caching is permitted, memory is configured as either write-back or write-through, as described in the [Chapter 3, “L1 and L2 Cache Operation.”](#)

6.6 Instruction Latency Summary

Instruction timing is shown in [Table 6-2](#) through [Table 6-8](#). The latency tables use the following conventions:

- Pipelined load/store and floating-point instructions are shown with cycles of total latency and throughput cycles separated by a colon.
- Floating-point instructions with a single entry in the cycles column are not pipelined.
- In addition, additional cycles due to serialization are indicated in the cycles column with the following:
 - e (execution serialization)
 - r (refetch serialization)
 - s (store serialization)

[Table 6-2](#) through [Table 6-8](#) list latencies associated with instructions executed by each execution unit. [Table 6-2](#) describes branch instruction latencies.

Table 6-2. Branch Operation Execution Latencies

Mnemonic	Primary	Extend	Form	Unit	Cycles
b[l][a]	18	—	i	BPU	1 ¹
bc[l][a]	16	—	b	BPU	1 ¹
bcctr[l]	19	528	xl	BPU	1 ¹
bclr[l]	19	016	xl	BPU	1, 2 ¹

¹ Branches that do not modify the LR or CTR can be folded and not dispatched. Branches that are dispatched go only to the CQ.

[Table 6-3](#) lists system operation instruction latencies.

Table 6-3. System Operation Instruction Execution Latencies

Mnemonic	Primary	Extend	Form	Unit	Cycles
eieio	31	854	X	LSU	3:5 {s}
isync	19	150	XL	— ¹	0{r}
mfmsr	31	083	X	IU2	3-2
mfspr (DBATs)	31	339	AFX	IU2	4:3{e}
mfspir (IBATs)	31	339	AFX	IU2	4:3
mfspir (MSS)	31	339	AFX	IU2	5{e} ²
mfspir (other)	31	339	AFX	IU2	3{e}
mfspir (time base)	31	339	AFX	IU2	5{e}

Table 6-3. System Operation Instruction Execution Latencies (continued)

Mnemonic	Primary	Extend	Form	Unit	Cycles
mf spr (VRSAVE)	31	339	AFX	IU2	3:2
mfsr	31	595	X	IU2	4:3
mfsrin	31	659	X	IU2	4:3
mftb	31	371	X	IU2	5{e}
mtmsr	31	146	X	IU2	2{e}
mtspr (DBATs)	31	467	AFX	IU2	2{e}
mtspr (IBATs)	31	467	AFX	IU2	2{e}
mtspr (MSS)	31	467	AFX	IU2	5{e}
mtspr (other)	31	467	AFX	IU2	2{e}
mtspr (XER)	31	467	AFX	IU2	2{e,r} ¹
mtsr	31	210	X	IU2	2{e}
mtsrin	31	242	X	IU2	2{e}
mttb	31	467	AFX	IU2	5{e}
rfi	19	050	XL	— ¹	0{r}
sc	17	— ¹	SC	— ¹	0{r}
sync	31	598	X	LSU	35 ³ {e,s}
tlbsync	31	566	X	LSU	3:5{s}

¹ Refetch serialized instructions (if marked with a 0-cycle execution time) do not have an execute stage, and all refetch serialized instructions have 1-cycle between the time they are completed and the time the target/sequential instruction enters the fetch1 stage.

² Core memory subsystem SPRs are implementation specific and are described in [Chapter 2, "Registers."](#)

³ Assuming a 5:1 processor to clock ratio.

Table 6-4 lists condition register logical instruction latencies.

Table 6-4. Condition Register Logical Execution Latencies

Mnemonic	Primary	Extend	Form	Unit	Cycles
crand	19	257	XL	IU2	2{e}
crandc	19	129	XL	IU2	2{e}
creqv	19	289	XL	IU2	2{e}
crnand	19	225	XL	IU2	2{e}
crnor	19	033	XL	IU2	2{e}
cror	19	449	XL	IU2	2{e}
crorc	19	417	XL	IU2	2{e}
crxor	19	193	XL	IU2	2{e}

Table 6-4. Condition Register Logical Execution Latencies (continued)

Mnemonic	Primary	Extend	Form	Unit	Cycles
mcrf	19	000	XL	IU2	2{e}
mcrxr	31	512	X	IU2	2{e}
mfcrr	31	019	X	IU2	2{e}
mctcrf	31	144	AFX	IU2/IU1	2{e}/1 ¹

¹ **mctcrf** of a single field is executed by an IU1 in a single cycle and is not serialized.

Table 6-5 lists integer unit instruction latencies.

Table 6-5. Integer Unit Execution Latencies

Mnemonic	Primary	Extend	Form	Unit	Cycles
addc[o][.]	31	010	XO	IU1	1
adde[o][.]	31	138	XO	IU1	1{e}
addi	14	—	D	IU1	1
addic	12	—	D	IU1	1
addic.	13	—	D	IU1	1
addis	15	—	D	IU1	1
addme[o][.]	31	234	XO	IU1	1{e}
addze[o][.]	31	202	XO	IU1	1{e}
add[o][.]	31	266	XO	IU1	1
andc[.]	31	060	X	IU1	1
andi.	28	—	D	IU1	1
andis.	29	—	D	IU1	1
and[.]	31	028	X	IU1	1
cmp	31	000	X	IU1	1
cmpi	11	—	D	IU1	1
cmpl	31	032	X	IU1	1
cmpli	10	—	D	IU1	1
cntlzw[.]	31	026	X	IU1	1
divwu[o][.] ¹	31	459	XO	IU2	23
divw[o][.] ²	31	491	XO	IU2	23
eqv[.]	31	284	X	IU1	1
extsb[.]	31	954	X	IU1	1 ³
extsh[.]	31	922	X	IU1	1 ³

Table 6-5. Integer Unit Execution Latencies (continued)

Mnemonic	Primary	Extend	Form	Unit	Cycles
mulhwu [.]	31	011	XO	IU2	4:2 ⁴
mulhw [.]	31	075	XO	IU2	4:2 ⁴
mulli	07	—	D	IU2	3:1
mull[o] [.]	31	235	XO	IU2	4:2 ⁴
nand [.]	31	476	X	IU1	1
neg[o] [.]	31	104	XO	IU1	1
nor [.]	31	124	X	IU1	1
orc [.]	31	412	X	IU1	1
ori	24	—	D	IU1	1
oris	25	—	D	IU1	1
or [.]	31	444	X	IU1	1
rlwimi [.]	20	—	M	IU1	1 ³
rlwinm [.]	21	—	M	IU1	1 ³
rlwnm [.]	23	—	M	IU1	1 ³
slw [.]	31	024	X	IU1	1 ³
srawi [.]	31	824	X	IU1	2 ⁵
sraw [.]	31	792	X	IU1	2 ⁵
srw [.]	31	536	X	IU1	1 ¹
subfc[o] [.]	31	008	XO	IU1	1
subfe[o] [.]	31	136	XO	IU1	1{e}
subfc	08	—	D	IU1	1
subfme[o] [.]	31	232	XO	IU1	1{e}
subfze[o] [.]	31	200	XO	IU1	1{e}
subf [.]	31	040	XO	IU1	1
tw	31	004	X	IU1	2
twi	03	—	D	IU1	2
xori	26	—	D	IU1	1
xoris	27	—	D	IU1	1
xor [.]	31	316	X	IU1	1

¹ For the special case of division by zero, the latency is 3 cycles.

² For the special case of division by zero or 0x8000_0000 divided by 0xFFFF_FFFF, the latency is 3 cycles

³ If the record bit is set, the result is available in 1 cycle and execution takes 2 cycles.

- ⁴ 32*32-bit multiplication has an early exit condition. If the 15 msbs of the B operand are either all set or all cleared, the multiply finishes with a latency of 3 and a throughput of 1.
- ⁵ **srawi[.]** and **sraw[.]** produce a GPR result in 1 cycle, but the full results, including the CA, OV, CR results, are available in 2 cycles.

Table 6-6 shows latencies for FPU instructions. Instructions with a single entry in the cycles column are not pipelined; all FPU stages are busy for the full duration of instruction execution and are unavailable to subsequent instructions. Floating-point arithmetic instructions execute in the FPU; floating-point loads and stores execute in the LSU.

For pipelined instructions, two numbers are shown separated by a colon. The first shows the number of cycles required to fill the pipeline. The second is the throughput once the pipeline is full. For example, **fabs[.]** passes through five stages with a 1-cycle throughput.

Table 6-6. Floating-Point Unit (FPU) Execution Latencies

Mnemonic	Primary	Extend	Form	Cycles
fabs[.]	63	264	X	5:1
fadds[.]	59	021	A	5:1
fadd[.]	63	021	A	5:1
fcmpo	63	032	X	5:1
fcmpu	63	000	X	5:1
fctiwz[.]	63	015	X	5:1
fctiw[.]	63	014	X	5:1
fdivs[.]	59	018	A	21
fdiv[.]	63	018	A	35
fmadds[.]	59	029	A	5:1
fmadd[.]	63	029	A	5:1
fmr[.]	63	072	X	5:1
fmsubs[.]	59	028	A	5:1
fmsub[.]	63	028	A	5:1
fmuls[.]	59	025	A	5:1
fmul[.]	63	025	A	5:1
fnabs[.]	63	136	X	5:1
fneg[.]	63	040	X	5:1
fnmadds[.]	59	031	A	5:1
fnmadd[.]	63	031	A	5:1
fnmsubs[.]	59	030	A	5:1
fnmsub[.]	63	030	A	5:1
fres[.]	59	024	A	14

Table 6-6. Floating-Point Unit (FPU) Execution Latencies (continued)

Mnemonic	Primary	Extend	Form	Cycles
frsp[.]	63	012	X	5:1
frsqste[.]	63	026	A	5:1
fsel[.]	63	023	A	5:1
fsubs[.]	59	020	A	5:1
fsub[.]	63	020	A	5:1
mcrfs	63	064	X	5(e)
mffs[.]	63	583	X	5(e)
mtfsb0[.]	63	070	X	5(e)
mtfsb1[.]	63	038	X	5(e)
mtfsfi[.]	63	134	X	5(e)
mtfsf[.]	63	711	XFL	5(e)

Table 6-7 shows load and store instruction latencies. Load/store multiple and string instruction cycles are represented as a fixed number of cycles plus a variable number of cycles, where n = the number of words accessed by the instruction. Pipelined load/store instructions are shown with total latency and throughput separated by a colon.

Table 6-7. Load/Store Unit (LSU) Instruction Latencies

Mnemonic	Primary	Extend	Form	Cycles ¹
dcba	31	758	X	3:1{s}
dcbf	31	86	X	3:11{s}
dcbst	31	54	X	3:11{s}
dcbt	31	278	X	3:1
dcbtst	31	246	X	3:1
dcbz	31	1014	X	3:1{s}
dss	31	582	X	3:1
dssall	31	582	X	3:1
dsts[t]	31	550	X	3:1
dst[t]	31	518	X	3:1
icbi	31	982	X	3:1{s}
lbz	34	—	D	3:1
lbzu	35	—	D	3:1
lbzux	31	119	X	3:1
lbzx	31	87	X	3:1

Table 6-7. Load/Store Unit (LSU) Instruction Latencies (continued)

Mnemonic	Primary	Extend	Form	Cycles ¹
lfd	50	—	D	4:1
lfdx	51	—	D	4:1
lfdux	31	631	X	4:1
lfdx	31	599	X	4:1
lfs	48	—	D	4:1
lfsu	49	—	D	4:1
lfsux	31	567	X	4:1
lfsx	31	535	X	4:1
lha	42	—	D	3:1
lhau	43	—	D	3:1
lhaux	31	375	X	3:1
lhax	31	343	X	3:1
lhbrx	31	790	X	3:1
lhz	40	—	D	3:1
lhzux	31	311	X	3:1
lhzx	31	279	X	3:1
lmw	46	—	D	3 + n
lswi	31	597	X	3 + n
lswx	31	533	X	3 + n
lvebx	31	7	X	3:1
lvehx	31	39	X	3:1
lvewx	31	71	X	3:1
lvsl	31	6	X	3:1
lvsl	31	38	X	3:1
lvx	31	103	X	3:1
lvxl	31	359	X	3:1
lwarx	31	20	X	3{e}
lwbrx	31	534	X	3:1
lwz	32	—	D	3:1
lwzu	33	—	D	3:1
lwzux	31	55	X	3:1
lwzx	31	23	X	3:1
stb	38	—	D	3:1{s}

Table 6-7. Load/Store Unit (LSU) Instruction Latencies (continued)

Mnemonic	Primary	Extend	Form	Cycles ¹
stbu	39	—	D	3:1{s}
stbux	31	247	X	3:1{s}
stbx	31	215	X	3:1{s}
stfd	54	—	D	3:3{s} ²
stfdu	55	—	D	3:3{s} ²
stfdux	31	759	X	3:3{s} ²
stfdx	31	727	X	3:3{s} ²
stfiwx	31	983	X	3:1{s}
stfs	52	—	D	3:3{s} ²
stfsu	53	—	D	3:3{s} ²
stfsux	31	695	X	3:3{s} ²
stfsx	31	663	X	3:3{s} ²
sth	44	—	D	3:1{s}
sthbrx	31	918	X	3:1{s}
sthu	45	—	D	3:1{s}
stmw	47	—	D	3 + n{s}
stswi	31	725	X	3+ n{s}
stswx	31	661	X	3 + n{s}
stvebx	31	135	X	3:1{s}
stvehx	31	167	X	3:1{s}
stviewx	31	199	X	3:1{s}
stvx	31	231	X	3:1{s}
stvxl	31	487	X	3:1{s}
stw	36	—	D	3:1{s}
stwbrx	31	662	X	3:1{s}
stwcx.	31	150	X	3:1{s}
stwu	37	—	D	3:1{s}
stwux	31	183	X	3:1{s}
stwx	31	151	X	3:1{s}
tlbie	31	306	X	3:1{s}
tlbld	31	978	X	3{e}
tlbli	31	1010	X	3{e}

- ¹ For cache operations, the first number indicates the latency for finishing a single instruction and the second number indicates the throughput for a large number of back-to-back cache operations. The throughput cycle may be larger than the initial latency because more cycles may be needed for the data to reach the cache. If the cache remains busy, subsequent cache operations cannot execute.
- ² Floating-point stores may take as many as 24 additional cycles if the value being stored is a denormalized number.

Table 6-8 describes AltiVec instruction latencies.

Table 6-8. AltiVec Instruction Latencies

Mnemonic	Primary	Extend	Form	Unit	Cycles ¹
mfvscr	04	1540	VX	VFPU	2{e}
mtvscr	04	1604	VX	VFPU	2{e}
vaddcuw	04	384	VX	VIU1	1
vaddfp	04	10	VX	VFPU	4:1
vaddsbs	04	768	VX	VIU1	1
vaddshs	04	832	VX	VIU1	1
vaddsws	04	896	VX	VIU1	1
vaddubm	04	0	VX	VIU1	1
vaddubs	04	512	VX	VIU1	1
vadduhm	04	64	VX	VIU1	1
vadduhs	04	576	VX	VIU1	1
vadduwm	04	128	VX	VIU1	1
vadduws	04	640	VX	VIU1	1
vand	04	1028	VX	VIU1	1
vandc	04	1092	VX	VIU1	1
vavgsh	04	1282	VX	VIU1	1
vavgsh	04	1346	VX	VIU1	1
vavgsw	04	1410	VX	VIU1	1
vavgub	04	1026	VX	VIU1	1
vavguh	04	1090	VX	VIU1	1
vavguw	04	1154	VX	VIU1	1
vcfsx	04	842	VX	VFPU	4:1
vcfux	04	778	VX	VFPU	4:1
vcmpbfp[.]	04	966 [1990]	VX	VFPU	2:1
vcmpeqfp[.]	04	198 [1222]	VX	VFPU	2:1
vcmpequb[.]	04	6 [1030]	VX	VIU1	1
vcmpequh[.]	04	70 [1094]	VX	VIU1	1

Table 6-8. AltiVec Instruction Latencies (continued)

Mnemonic	Primary	Extend	Form	Unit	Cycles ¹
vcmpequw [.]	04	134 [1158]	VX	VIU1	1
vcmpgefp [.]	04	454 [1478]	VX	VFPU	2:1
vcmpgtfp [.]	04	710 [1734]	VX	VFPU	2:1
vcmpgtsb [.]	04	774 [1798]	VX	VIU1	1
vcmpgtsh [.]	04	838 [1862]	VX	VIU1	1
vcmpgtsw [.]	04	902 [1926]	VX	VIU1	1
vcmpgtub [.]	04	518 [1542]	VX	VIU1	1
vcmpgtuh [.]	04	582 [1606]	VX	VIU1	1
vcmpgtuw [.]	04	646 [1670]	VX	VIU1	1
vctxs	04	970	VX	VFPU	4:1
vctuxs	04	906	VX	VFPU	4:1
vexptefp	04	394	VX	VFPU	4:1
vlogefp	04	458	VX	VFPU	4:1
vmaddfp	04	46	VA	VFPU	4:1
vmaxfp	04	1034	VX	VFPU	2:1
vmaxsb	04	258	VX	VIU1	1
vmaxsh	04	322	VX	VIU1	1
vmaxsw	04	386	VX	VIU1	1
vmaxub	04	2	VX	VIU1	1
vmaxuh	04	66	VX	VIU1	1
vmaxuw	04	130	VX	VIU1	1
vmhaddshs	04	32	VA	VIU2	4:1
vmhraddshs	04	33	VA	VIU2	4:1
vminfp	04	1098	VX	VFPU	2:1
vminsb	04	770	VX	VIU1	1
vminsh	04	834	VX	VIU1	1
vminsw	04	898	VX	VIU1	1
vminub	04	514	VX	VIU1	1
vminuh	04	578	VX	VIU1	1
vminuw	04	642	VX	VIU1	1
vmladduhm	04	34	VA	VIU2	4:1
vmrghb	04	12	VX	VPU	2:1
vmrghh	04	76	VX	VPU	2:1

Table 6-8. AltiVec Instruction Latencies (continued)

Mnemonic	Primary	Extend	Form	Unit	Cycles ¹
vmrghw	04	140	VX	VPU	2:1
vmrglb	04	268	VX	VPU	2:1
vmrglh	04	332	VX	VPU	2:1
vmrglw	04	396	VX	VPU	2:1
vmsummbm	04	37	VA	VIU2	4:1
vmsumshm	04	40	VA	VIU2	4:1
vmsumshs	04	41	VA	VIU2	4:1
vmsumubm	04	36	VA	VIU2	4:1
vmsumuhm	04	38	VA	VIU2	4:1
vmsumuhs	04	39	VA	VIU2	4:1
vmulesb	04	776	VX	VIU2	4:1
vmulesh	04	840	VX	VIU2	4:1
vmuleub	04	520	VX	VIU2	4:1
vmuleuh	04	584	VX	VIU2	4:1
vmulosb	04	264	VX	VIU2	4:1
vmulosh	04	328	VX	VIU2	4:1
vmuloub	04	8	VX	VIU2	4:1
vmulouh	04	72	VX	VIU2	4:1
vnmsubfp	04	47	VA	VFPU	4:1
vnor	04	1284	VX	VIU1	1
vor	04	1156	VX	VIU1	1
vperm	04	43	VA	VPU	2:1
vpkpx	04	782	VX	VPU	2:1
vpkshss	04	398	VX	VPU	2:1
vpkshus	04	270	VX	VPU	2:1
vpkswss	04	462	VX	VPU	2:1
vpkswus	04	334	VX	VPU	2:1
vpkuhum	04	14	VX	VPU	2:1
vpkuhus	04	142	VX	VPU	2:1
vpkuwum	04	78	VX	VPU	2:1
vpkuwus	04	206	VX	VPU	2:1
vrefp	04	266	VX	VFPU	4:1
vrfim	04	714	VX	VFPU	4:1

Table 6-8. AltiVec Instruction Latencies (continued)

Mnemonic	Primary	Extend	Form	Unit	Cycles ¹
vrfn	04	522	VX	VFPU	4:1
vrfip	04	650	VX	VFPU	4:1
vrfiz	04	586	VX	VFPU	4:1
vrlb	04	4	VX	VIU1	1
vrlh	04	68	VX	VIU1	1
vrlw	04	132	VX	VIU1	1
vrsqrtefp	04	330	VX	VFPU	4:1
vsel	04	42	VA	VIU1	1
vsl	04	452	VX	VPU	2:1
vsib	04	260	VX	VIU1	1
vsldoi	04	44	VA	VPU	2:1
vslh	04	324	VX	VIU1	1
vslo	04	1036	VX	VPU	2:1
vslw	04	388	VX	VIU1	1
vspltb	04	524	VX	VPU	2:1
vsplth	04	588	VX	VPU	2:1
vspltisb	04	780	VX	VPU	2:1
vspltish	04	844	VX	VPU	2:1
vspltisw	04	908	VX	VPU	2:1
vspltw	04	652	VX	VPU	2:1
vsr	04	708	VX	VPU	2:1
vsrab	04	772	VX	VIU1	1
vsrah	04	836	VX	VIU1	1
vsraw	04	900	VX	VIU1	1
vsrb	04	516	VX	VIU1	1
vsrh	04	580	VX	VIU1	1
vsro	04	1100	VX	VPU	2:1
vsrw	04	644	VX	VIU1	1
vsubcuw	04	1408	VX	VIU1	1
vsubfp	04	74	VX	VFPU	4:1
vsubsubs	04	1792	VX	VIU1	1
vsubshs	04	1856	VX	VIU1	1
vsubsws	04	1920	VX	VIU1	1

Table 6-8. AltiVec Instruction Latencies (continued)

Mnemonic	Primary	Extend	Form	Unit	Cycles ¹
vsububm	04	1024	VX	VIU1	1
vsububs	04	1536	VX	VIU1	1
vsubuhm	04	1088	VX	VIU1	1
vsubuhs	04	1600	VX	VIU1	1
vsubuwm	04	1152	VX	VIU1	1
vsubuws	04	1664	VX	VIU1	1
vsum2sws	04	1672	VX	VIU2	4:1
vsum4sbs	04	1800	VX	VIU2	4:1
vsum4shs	04	1608	VX	VIU2	4:1
vsum4ubs	04	1544	VX	VIU2	4:1
vsumsws	04	1928	VX	VIU2	4:1
vupkhp	04	846	VX	VPU	2:1
vupkhsb	04	526	VX	VPU	2:1
vupkhsh	04	590	VX	VPU	2:1
vupklp	04	974	VX	VPU	2:1
vupklsb	04	654	VX	VPU	2:1
vupklsh	04	718	VX	VPU	2:1
vxor	04	1220	VX	VIU1	1

¹ Most AltiVec floating-point instructions on the e600 core (regardless of Java/non-Java mode) have a 4-cycle latency. However, some VFPU instructions have a 2-cycle latency, which under some conditions may cause other instructions to have a greater than 4-cycle latency (see [Section 6.4.5.1.4, “Vector Floating-Point Unit \(VFPU\) Execution Timing,”](#) for details).

6.7 Instruction Scheduling Guidelines

This section provides an overview of instruction scheduling guidelines, followed by detailed examples showing how to optimize scheduling with respect to various pipeline stages. Performance can be improved by avoiding resource conflicts and scheduling instructions to take fullest advantage of the parallel execution units. Instruction scheduling can be improved by observing the following guidelines:

- To reduce branch mispredictions, separate the instruction that sets CR bits from the branch instruction that evaluates them. Because there can be no more than 24 instructions in the processor (with the instruction that sets CR in CQ0 and the dependent branch instruction executing in IQ7), there is no advantage to having more than 22 instructions between them.
- Likewise, when branching to a location specified by the CTR or LR, separate the **mtspr** instruction that initializes the CTR or LR from the dependent branch instruction. This ensures the register values are immediately available to the branch instruction.

- Schedule instructions so three can be dispatched at a time.
- Schedule instructions to minimize stalls due to busy execution units. To avoid branch stalls, the e600 core has three branch prediction buffers. This allows the branch prediction engine to go deeper before stalling.
- Avoid scheduling high-latency instructions close together. Interspersing single-cycle latency instructions between longer-latency instructions minimizes the effect that instructions such as integer divide and multiply can have on throughput.
- Avoid using serializing instructions.
- Schedule instructions to avoid dispatch stalls:
 - Intersperse instructions to maximize the ability to dispatch three GPR instructions, two VR instructions, and one FPR instruction.
 - 16 instructions can be tracked in the CQ; therefore, 16 instructions can be in the execute stages at any one time
 - There are 16 GPR rename registers, so only 16 GPRs can be specified as destination operands at any time. If no rename registers are available, instructions cannot enter the execute stage and remain in the reservation station or IQ until they become available.
Note that load with update address instructions use two destination registers
 - Similarly, there are 16 FPR rename registers and 16 VR rename registers, so 16 FPR and 16 AltiVec rename register destination operands can be in the execute and complete stages at any time.
- Avoid branches where possible; favor fall-through branches over taken branches.

The following sections give detailed information regarding optimizing code for the e600 core pipeline stages.

6.7.1 Fetch/Branch Considerations

The following is a list of branch instructions and the resources required to avoid stalling the fetch unit in the course of branch resolution:

- The **bclr** instruction requires LR availability for resolution. However, it uses the link stack to predict the target address in order to avoid stalling fetch.
- The **bcctr** instruction requires CTR availability.
- The branch conditional on counter decrement and the CR condition requires CTR availability or the CR condition must be false.
- A fourth conditional branch instruction cannot be executed following three unresolved predicted branch instructions.

6.7.1.1 Fetching Examples

Branches that target an instruction at or near the end of a cache block can cause instruction supply problems. Consider a four-instruction loop branch (including the branch) where the entry point is the last word of the cache block. The e600 core needs at least 2 cycles to fetch the four instructions because the

cache block boundary breaks the four instructions into two accesses. Aligning this loop significantly increases the instruction supply.

Additionally, on the e600 core this tight loop encounters the branch-taken bubble problem. That is, the BTIC supplies instructions 1 cycle after the branch executes. For the instructions in the cache block crossing case, this leads to four instructions fetched every 3 cycles. Aligning instructions to be within a cache block increases the number of instructions fetched to 4 every 2 cycles. For longer loops, the branch-taken bubble overhead can be better amortized or in some cases can disappear (because the branch is executed early and the bubble disappears at dispatch time). Software loop unrolling can increase the number of instructions per branch.

NOTE

The BTIC contains targets for only **b** and **bc** branches. Indirect branches (**bcctr** and **bclr**) must go to the instruction cache for instructions, which incurs an additional cycle of fetch latency (branch-taken bubble).

6.7.1.1.1 Fetch Alignment Example

The following code loop is a simple array accumulation operation.

```

xxxxxx18 loop:  lwzu r10,0x4(r9)
xxxxxx1C          add r11,r11,r10
xxxxxx20          bdnz loop
    
```

The **lwzu** and **add** are the last two instructions in one cache block, and the **bdnz** is the first instruction in the next. In this example, the fetch supply is the primary restriction. Table 6-9 assumes instruction cache and BTIC hits. The **lwzu/add** of the second iteration are available for dispatch in cycle 3, as a result of a BTIC hit for the **bdnz** executed in cycle 1. The **bdnz** of the second iteration is available in the IQ one cycle later (cycle 4) because the cache block break forced a fetch from the instruction cache. Overall, the loop is limited to one iteration for every 3 cycles.

Table 6-9. Fetch Alignment Example

Instruction	0	1	2	3	4	5	6	7	8	9	10	11
lwzu (1)	D	I	E0	E1	E2	C						
add (1)	D	I	-	-	-	E	C					
bdnz (1)	F2	BE	D	-	-	-	C					
lwzu (2)				D	I	E0	E1	E2	C			
add (2)				D	I	-	-	-	E	C		
bdnz (2)			F1	F2	BE	D	-	-	-	C		
lwzu (3)							D	I	E0	E1	E2	C
add (3)							D	I	-	-	-	E
bdnz (3)						F1	F2	BE	D	-	-	-

Performance can be increased if the loop is aligned so that all three instructions are in the same cache block, as in the following example.

Instruction Timing

```

xxxxxx00 loop:  lwzu r10,0x4(r9)
xxxxxx04          add r11,r11,r10
xxxxxx08          bdnz loop
  
```

The fact that the loop fits in the same cache block allows the BTIC entry to provide all three instructions. [Table 6-10](#) shows pipelined execution results (again assuming BTIC and instruction cache hits). While fetch supply is still a bottleneck, it is improved by proper alignment. The loop is now limited to one iteration every 2 cycles, increasing performance by 50%.

Table 6-10. Loop Example—Three Iterations

Instruction	0	1	2	3	4	5	6	7	8	9
lwzu (1)	D	I	E0	E1	E2	C				
add (1)	D	I	-	-	-	E	C			
bdnz (1)	BE	D	-	-	-	-	C			
lwzu (2)			D	I	E0	E1	E2	C		
add (2)			D	I	-	-	-	E	C	
bdnz (2)			BE	D	-	-	-	-	C	
lwzu (3)					D	I	E0	E1	E2	C
add (3)					D	I	-	-	-	E
bdnz (3)					BE	D	-	-	-	-

Loop unrolling and vectorization can further increase performance.

6.7.1.1.2 Branch-Taken Bubble Example

The following code shows how favoring taken branches affects fetch supply.

```

xxxxxx00          lwz r10,0x4(r9)
xxxxxx04          cmpi 4,r10,0x0
xxxxxx08          bne 4, targ
xxxxxx0C          stw r11,0x4(r9)
xxxxxx10 targ    add (next basic block)
  
```

This example assumes the **bne** is usually taken (that is, most of the data in the array is non-zero). [Table 6-11](#) assumes correct prediction of the **bne**, and cache and BTIC hits.

Table 6-11. Branch-Taken Bubble Example

Instruction	0	1	2	3	4	5	6
lwz	D	I	E0	E1	E2	C	
cmpi	D	I	-	-	-	E	C
bne	BE						
add			D	I	E	-	C

Rearranging the code as follows improves the fetch supply.

```

xxxxxx00      lwz r10,0x4(r9)
xxxxxx04      cmpi 4,r10,0x0
xxxxxx08      beq 4,targ
xxxxxx0C targ2 add (next basic block)
...
yyyyyy00 targ stw r11,0x4(r9)
yyyyyy04      b targ2
    
```

Using the same assumptions as before, [Table 6-12](#) shows the performance improvement. Note that the first instruction of the next basic block (**add**) completes in the same cycle as before. However, by avoiding the branch-taken bubble (because the branch is usually not taken), it also dispatches 1 cycle earlier, so that the next basic block begins executing 1 cycle sooner.

Table 6-12. Eliminating the Branch-Taken Bubble

Instruction	0	1	2	3	4	5	6	7	8	9	10
lwz	D	I	E0	E1	E2	C					
cmpi	D	I	-	-	-	E	C				
beq	BE	D	-	-	-	-	C				
add		D	I	E	-	-	C				

6.7.1.2 Branch Conditionals

The cost of mispredictions increases with pipeline length. The following section shows common problems and suggests how to minimize them.

6.7.1.2.1 Branch Mispredict Example

[Table 6-13](#) uses the same code as the two previous examples but assumes the **bne** mispredicts. The **cmpi** executes in cycle 5, which means the branch mispredicts in cycle 6 and the fetch pipeline restarts at that correct target for the **add** in cycle 7. This particular mispredict effectively costs 7 cycles (**add** dispatches in cycle 2 in [Table 6-11](#) and in cycle 9 in [Table 6-13](#)).

Table 6-13. Misprediction Example

Instruction	0	1	2	3	4	5	6	7	8	9	10	11	12
lwz	D	I	E0	E1	E2	C							
cmpi	D	I	-	-	-	E	C						
bne	BE						M						
add								F1	F2	D	I	E	C

6.7.1.2.2 Branch Loop Example

Use CTR whenever possible for branch loops, especially for tight inner loops. After the CTR is loaded (using **mtctr**), a branch dependent on the CTR requires no directional prediction. Additionally, loop termination conditions are always handled correctly, which is not so with the normal branch predictor.

Instruction Timing

```

xxxxxx18 outer_loop:addi. r6,r6,#FFFF
xxxxxx1C          cmpi 1,r6,#0
xxxxxx20 inner_loop:addic. r7,r7,#FFFF
xxxxxx24          lwzu r10,0x4(r9)
xxxxxx28          add r11,r11,r10
xxxxxx2C          bne inner_loop
xxxxxx30          stwu r11,0x4(r8)
xxxxxx34          xor r11,r11,r11
xxxxxx38          ori r7,r0,#4
xxxxxx3C          bne cr1,outer_loop
  
```

For this example, assume the inner loop executes four times per outer iteration. The inner loop termination is always mispredicted because the branch predictor learns to predict the inner **bne** as taken, which is wrong every fourth time. [Table 6-14](#) shows that the misprediction causes the outer loop code to be dispatched in cycle 13. If the branch had been correctly predicted as not taken, these instructions would have dispatched 5 cycles earlier in cycle 8.

Table 6-14. Three Iterations of Code Loop

Instruction	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
addi	D	I	E	C											
cmp	D	I	-	E	C										
addic (1)	F2	D	I	E	C										
lwzu (1)	F2	D	I	E0	E1	E2	C								
add (1)	F2	D	I	-	-	-	E	C							
bne (1)	F2	BE													
addic. (2)				D	I	E	-	C							
lwzu (2)				D	I	E0	E1	E2	C						
add (2)				D	I	-	-	-	E	C					
bne (2)				BE											
addic. (3)						D	I	E	-	C					
lwzu (3)						D	I	E0	E1	E2	C				
add (3)						D	I	-	-	-	E	C			
bne (3)						BE									
addic. (4)								D	I	E	-	C			
lwzu (4)								D	I	E0	E1	E2	C		
add (4)								D	I	-	-	-	E	C	
bne (4)								BE			M				
stwu												F1	F2	D	I
xor												F1	F2	D	I
ori												F1	F2	D	I
bne												F1	F2	BE	

The following code uses CTR, which shortens the loop because the compare test (done by the **addic.** at `xxxxxx20`) is combined into the **bdnz** branch.

```

xxxxxx1C outer_loop:addic. r6,r6,#FFFF
xxxxxx20 inner_loop:lwzu r10,0x4(r9)
xxxxxx24         add r11,r11,r10
xxxxxx28         bdnz inner_loop
xxxxxx2C         mtctr r7
xxxxxx30         stwu r11,0x4(r8)
xxxxxx34         xor r11,r11,r11
xxxxxx38         bne 0,outer_loop
    
```

As [Table 6-15](#) shows, the inner loop termination branch does not need to be predicted and is executed as a fall-through branch. Instructions in the outer loop start dispatching in cycle 8, saving 5 cycles over the code in [Table 6-14](#). Note that because **mtctr** is execution serialized, it does not complete until cycle 16; nevertheless, the CTR value is forwarded to the BPU by cycle 11. This early forwarding starts for a **mtctr/mlr** when the instruction reaches reservation station 0 of the IU2 and the source register for the **mtctr/mlr** is available.

Table 6-15. Code Loop Example Using CTR

Instruction	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
addic	D	I	E	C														
lwzu (1)	F2	D	I	E0	E1	E2	C											
add (1)	F2	D	I	-	-	-	E	C										
bdnz (1)	F2	BE	D	-	-	-	-	C										
lwzu (2)				D	I	E0	E1	E2	C									
add (2)				D	I	-	-	-	E	C								
bdnz (2)				BE	D	-	-	-	-	C								
lwzu (3)						D	I	E0	E1	E2	C							
add (3)						D	I	-	-	-	E	C						
bdnz (3)						BE	D	-	-	-	-	C						
lwzu (4)								D	I	E0	E1	E2	C					
add (4)								D	I	-	-	-	E	C				
bdnz (4)								BE	D	-	-	-	-	C				
mtctr									D	I						E	C	
stwu									D	I	E0	-	-	-	-	-	-	C
xor									-	D	I	E	-	-	-	-	-	C
bne									BE									

6.7.1.3 Static versus Dynamic Prediction

Using static prediction (`HID0[BHT] = 0`) means that the hint bit in the branch opcode predicts the branch and the dynamic predictor (the BHT) is ignored. Sometimes static prediction is superior, either through

informed guessing or through available profile-directed feedback. Run-time for code using static prediction is more nearly deterministic, which can be useful in an embedded system.

6.7.1.4 Using the Link Stack for Branch Indirect

On the e600 core, a **bclr** uses the link stack to predict the target. To use the link stack correctly, each branch-and-link instruction must be paired with a branch-to-link instruction. Using the architecture-defined LR for computed targets corrupts the link stack. In general, the CTR should be used for computed target addresses and the LR should be used only for call/return addresses. If using the CTR for a loop conflicts with a computed GOTO, the computed GOTO should be used and the loop should be converted to a GPR form.

When generating position-independent code, many compilers use an instruction sequence such as the following to obtain the current instruction address.

```
bcl 20,31,$+4
mflr r3
```

Note that this is not a true call and is not paired with a return. The link stack is optimized to ignore position-independent code where the **bcl 20,31,\$+4** form is used. This conditional call, which is used only for putting the instruction address in a program-visible register, does not force a push on the link stack and is treated as a not-taken branch.

6.7.1.4.1 Link Stack Example

The following code sequence is a common code sequence for a subroutine call/return sequence, where main calls foo, foo calls ack, and ack possibly calls additional functions (not shown).

```
main:    ...
        mflr  r5
        stwu r5,-4(r1)
        bl   foo
5       add  r3,r3,r20
        ....
foo:    stwu r31,-4(r1)
        stwu r30,-4(r1)
        ....
        mflr  r4
        stwu r4,-4(r1)
        bl   ack
        add  r3,r3,r6
        ....
0       lwzu r30,4(r1)
1       lwzu r31,4(r1)
2       lwzu r5,4(r1)
3       mtlr r5
4       bclr
ack:    ....
        (possible calls to other functions)
        ....
        lwzu r4,4(r1)
        mtlr r4
        bclr
```


The **bl** in main pushes a value onto the hardware-managed link stack (and to the architecturally-defined LR). Then, the **bl** in foo pushes a second value onto the stack.

When ack later returns through the **bclr**, the hardware link stack is used to predict the value of the LR, if the actual value of the LR is not available when the branch is executed (typically because the **lwzu/mtlr** pair has not finished executing). It also pops a value off of the stack, leaving only the first value on the stack. This occurs again with the **bclr** in foo, which returns to main, leaving the stack empty.

Table 6-16 shows the performance implications of the link stack. The following code starts executing from instruction 0 in procedure foo.

Table 6-16. Link Stack Example

Inst#	Instruction	0	1	2	3	4	5	6	7	8	9	10	11	12
0	lwzu r30, 4(r1)	F1	F2	D	I	E0	E1	E2	C					
1	lwzu r31, 4(r1)	F1	F2	-	D	I	E0	E1	E2	C				
2	lwzu r5, 4(r1)	F1	F2	-	-	D	I	E0	E1	E2	C			
3	mtlr		F1	F2	-	D	I	-	-	-	*	-	E	C
4	bclr		F1	F2	BE	D								
...														
5	add r3,r3,r20					F1	F2	D	I	E	-	-	-	C

With link stack prediction, the BPU can successfully predict the target of the **bclr** (instruction 4), which allows the instruction to be executed at the return address (instruction 5) in cycle 8. The IU2 forwards the LR value to the BPU in cycle 9 (which implies that the branch resolution occurs in cycle 10), even though it cannot execute from an execution serialization viewpoint until cycle 11.

Without link stack prediction, the branch would stall on the LR dependency and not execute until after the LR is forwarded (that is, branch execution would occur in cycle 10), which allows instruction 5 not to execute until cycle 15 (seven cycles later than it executes with link stack prediction).

6.7.1.4.2 Position-Independent Code Example

Position-independent code is used when not all addresses are known at compile time or link time. Because performance is typically not good, position-independent code should be avoided when possible. The following example expands on the code sequence, which is described in the section “Conditional Branch Control,” in the *Programming Environments Manual*.

Table 6-17. Position-Independent Code Example

Inst#	Instruction	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
0	bcl 20, 31, \$+4	F1	F2	BE	D	C													
1	mflr r2	F1	F2	-	D	I	-	E0	E1	E2	E3	F	C						
2	addi r2, r2, #constant	F1	F2	-	D	I	-	-	-	-	-	E	C						
3	mtctr r2	F1	F2	-	-	D	I	-	-	-	-	-	*	-	E	C			

Table 6-17. Position-Independent Code Example (continued)

Inst#	Instruction	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
4	bcctr		F1	F2	-	-	-	-	-	-	-	-	-	BE					
...																			
5	add r3, r3, r20														F1	F2	D	I	E

Because a return (**bclr**) is never paired with this **bcl** (instruction 0), the e600 core takes two special actions when it recognizes this special form (“**bcl** 20,31,\$+4”):

- Although the **bcl** does update the link register as architecturally required, it does not push the value onto the link stack. Not pairing a return with this **bcl** prevents the link stack from being corrupted, which would likely require a later branch mispredict for some later **bclr**.
- Because the branch has the same next instruction address whether it is taken or fall-through, the branch is forced as a fall-through branch. This avoids a potential branch-taken bubble and saves a cycle.

The instruction address is available for executing a subsequent operation (instruction 2, **addi**) in cycle 10, primarily due to the long latency of the execution-serialized **mflr**. However, the data must be transferred back to the BPU through the CTR, which prevents the **bcctr** from executing until cycle 12, so its target instruction (5) cannot start execution until cycle 17.

Note that instructions 3 and 4 must be a **mtctr/bcctr** pair rather than a **mtlr/bclr** pair. A **bclr** would try to use the link stack to predict the target address, which would almost certainly be an address mispredict, which would be even more costly than the 7-cycle branch execution stall for instruction 4 in this example. In addition, an address mispredict would require the link stack to be flushed, which would mean that **bclr** instructions later in the program would stall rather than use link stack address prediction, further degrading performance.

6.7.1.5 Branch Folding

Branches that do not set the LR or update the CTR are eligible for folding. Taken branches are folded immediately. In the e600 core, not-taken branches cannot be fall-through folded if they are in IQ0–IQ2 (the dispatch entries); however, branches are removed in the cycle after execution if they are in IQ3–IQ7.

6.7.2 Dispatch Unit Resource Requirements

The following is a list of resources required to avoid stalls in the dispatch unit:

- The appropriate issue queue must be available.
- Sufficient CQ entries must be available.
- Previous instructions in the IQ dispatch entries (IQ0–IQ2) must dispatch.
- Needed rename registers must be available.

The following sections describe how to optimize code for dispatch.

6.7.2.1 Dispatch Groupings

Maximum dispatch throughput is three instructions per cycle. The dispatch process includes a CQ available check, an issue queue available check, a branch ready check, and a rename check.

The dispatcher can send three instructions to the various issue queues, with a maximum of three to the GIQ, two to the VIQ, and one to the FIQ. Thus only two instructions can be dispatched per cycle to the AltiVec units (VIU1, VIU2, VPU, and VFPU). Only one FPU instruction can be dispatched per cycle, so three **fadds** would take three cycles to dispatch.

Only one load/store instruction can dispatch per cycle.

The dispatcher can rename as many as four GPRs, three VRs, and two FPRs per cycle, so a three-instruction dispatch window composed of **vaddfp**, **vaddfp**, and **lviewx** could be dispatched in one cycle.

Note that a load/store update form (for example, **lwzu**), requires a GPR rename for the update. This means that an **lwzu** needs two GPR renames and an **lfd** needs one FPU rename and one GPR rename. The possibility that one instruction may need two GPR renames means that even though the e600 core has a 16-entry CQ and 16 GPR renames, GPR renames could run out even though there is space in the CQ, as when eight **lwzu** instructions are in the CQ. Eight CQ entries are available, but because all 16 GPR renames are in use, no instruction needing a GPR target can be dispatched.

The restriction to four GPR renames in a dispatch group means that the sequence **lwzu**, **add**, **add** can be dispatched in one cycle. The instruction pair **lwzu**, **lwzu** also uses four GPR renames and passes this rule but is disallowed by the rule that enforces a dispatch of only one load/store per cycle.

6.7.2.1.1 Dispatch Stall Due to Rename Availability

Table 6-18 shows a code example that allows dispatch stall due to rename availability:

Table 6-18. Dispatch Stall Due to Rename Availability

Inst#	Instruction	0	1	2	3	4	5	6	7	8	9	...	25	26	27	28	29	30
0	divw r4,r3,r2	F1	F2	D	I	E0	E1	E2	E3	E4	E5	...	E21	E22	C	WB		
1	lwzu r22,0x04(r1)	F1	F2	D	I	E0	E1	E2	-	-	-	...	-	-	C	WB		
2	lwzu r23,0x04(r1)	F1	F2	-	D	I	E0	E1	E2	-	-	...	-	-	-	C	WB	
3	lwzu r24,0x04(r1)	F1	F2	-	-	D	I	E0	E1	E2	-	...	-	-	-	-	C	WB
4	lwzu r25,0x04(r1)		F1	F2	-	-	D	I	E0	E1	E2	...	-	-	-	-	-	C
5	lwzu r26,0x04(r1)		F1	F2	-	-	-	D	I	E0	E1	...	-	-	-	-	-	
6	lwzu r27,0x04(r1)		F1	F2	-	-	-	-	D	I	E0	...	-	-	-	-	-	
7	lwzu r28,0x04(r1)		F1	F2	-	-	-	-	-	D	I	...	-	-	-	-	-	
8	lwzu r29,0x04(r1)			F1	F2	-	-	-	-	-	-	...	-	-	-	-	D	I

Instruction 8 stalls in cycle 9 because it needs 2 renames, and 15 renames are in use (1 for the **divw**, and 2 each for instructions 1–7). Since only 16 GPR renames are allowed, instruction 8 cannot be dispatched until at least one rename is released.

When the **div** later completes (cycle 27 in example above), renames are released during the write-back stage and instruction 8 can thus dispatch in cycle 29.

6.7.2.2 Dispatching Load/Store Strings and Multiples

The e600 core splits multiples (**lmw** and **stmw**) and strings (**lsw*** and **stsw***) into micro-operations at the dispatch point. The processor can dispatch only one micro-operation per cycle, which does not use the dispatcher to its full advantage.

Using load/store multiple instructions is best restricted to cases where minimizing code size is critical or where there are no other available instructions to be scheduled, such that the under-utilization of the dispatcher is not a consideration.

6.7.2.2.1 Example of Load/Store Multiple Micro-Operation Generation

Consider the following assembly code:

```

0 lmw r25,0x00(r1)
1 addi r25,r25,0x01
2 addi r26,r26,0x01
3 addi r27,r27,0x01
4 addi r28,r28,0x01
5 addi r29,r29,0x01
6 addi r30,r30,0x01
7 addi r31,r31,0x01
    
```

The load multiple specified with value 25 loads registers 25–31. The e600 core splits this instruction into seven micro-operations at dispatch, after which the **lmw** executes as multiple operations, as shown in [Table 6-19](#).

Table 6-19. Load/Store Multiple Micro-Operation Generation Example

Inst#	Instruction	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0-0	lmw r25,0x00(r1)	F1	F2	D	I	E0	E1	E2	C								
0-1	lmw r26,0x04(r1)	F1	F2	-	D	I	E0	E1	E2	C							
0-2	lmw r27,0x08(r1)	F1	F2	-	-	D	I	E0	E1	E2	C						
0-3	lmw r28,0x0C(r1)	F1	F2	-	-	-	D	I	E0	E1	E2	C					
0-4	lmw r29,0x10(r1)	F1	F2	-	-	-	-	D	I	E0	E1	E2	C				
0-5	lmw r30,0x14(r1)	F1	F2	-	-	-	-	-	D	I	E0	E1	E2	C			
0-6	lmw r31,0x1C(r1)	F1	F2	-	-	-	-	-	-	D	I	E0	E1	E2	C		
1	addi r25,r25,0x01	F1	F2	-	-	-	-	-	-	D	I	E	-	-	C		
2	addi r26,r26,0x01	F1	F2	-	-	-	-	-	-	D	I	E	-	-	C		
3	addi r27,r27,0x01	F1	F2	-	-	-	-	-	-	-	D	I	E	-	-	C	
4	addi r28,r28,0x01		F1	F2	-	-	-	-	-	-	D	I	E	-	-	C	
5	addi r29,r29,0x01		F1	F2	-	-	-	-	-	-	D	I	E	-	-	C	

Table 6-19. Load/Store Multiple Micro-Operation Generation Example (continued)

Inst#	Instruction	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
6	addi r30,r30,0x01		F1	F2	-	-	-	-	-	-	-	D	I	E	-	-	C
7	addi r31,r31,0x01		F1	F2	-	-	-	-	-	-	-	D	I	-	E	-	C

Because the e600 core can dispatch only one LSU operation per cycle, the **lmw** is micro-oped at a rate of 1 per cycle and so in this example takes 7 cycles to dispatch all the operations. However, when the last operation in the multiple is dispatched (cycle 8), instructions 1 and 2 can dispatch along with it.

The use of load/store string instructions is strongly discouraged.

6.7.3 Issue Queue Resource Requirements

Instructions cannot be issued unless the specified execution unit is available. The following sections describe how to optimize use of the three issue queues.

6.7.3.1 GPR Issue Queue (GIQ)

As many as three instructions can be dispatched to the six-entry GPR issue queue (GIQ) per cycle. As many as three instructions can be issued in any order to the LSU, IU2, and IU1 reservation stations from the bottom three GIQ entries.

Issuing instructions out of order can help in a number of situations. For example, if the IU2 is busy and a multiply is stalled at the bottom GIQ entry, instructions in the next two GIQ entries can be issued to LSU or IU1s, bypassing that multiply.

The following sequence is not well scheduled, but effectively the e600 core dynamically reschedules around the potential multiply bottleneck.

```

0      xxxxxx00      mulhw r10,r20,r21
1      xxxxxx04      mulhw r11,r22,r23
2      xxxxxx08      mulhw r12,r24,r25
3      xxxxxx0C      lwzu r13,0x4(r9)
4      xxxxxx10      add r10,r10,r11
5      xxxxxx14      add r13,r13,r25
6      xxxxxx18      add r14,r5,r4
7      xxxxxx20      subf r15,r6,r4
    
```

The timing for this sequence, illustrated in [Table 6-20](#), shows which instructions are in which GIQ entries. Instruction 3 issues out-of-order in cycle 2; instructions 4 and 5 issue out-of-order in cycle 3.

Note that instruction 7 (**subf**) does not issue in cycle 4 because all three IU1 reservation stations have an instruction (4, 5, and 6). Instructions 4 and 5 are waiting in the reservation station for their source registers to be forwarded from the IU2 and LSU, respectively. Because instruction 6 executes immediately after issue (in cycle 5), instruction 7 can issue in that cycle.

Table 6-20. GIQ Timing Example

Inst#	Instruction	0	1	2	3	4	5	6	7	8	9	10	11
0	mulhw	D	I	E0	E0	E1	F	C					
1	mulhw	D	—	I	-	E0	E0	E1	F	C			
2	mulhw	D	—	—	—	I	—	E0	E0	E1	F	C	
3	lwzu	-	D	I	E0	E1	E2	—	—	—	—	C	
4	add	F2	D	—	I	—	—	—	E	—	—	C	
5	add	F2	D	—	I	—	—	E	—	—	—	—	C
6	add	F2	—	D	—	I	E	—	—	—	—	—	C
7	subf	F2	—	—	D	—	I	E	—	—	—	—	C

GIQ5													
GIQ4		5											
GIQ3		4	6										
GIQ2	2	3	5	7									
GIQ1	1	2	4	6									
GIQ0	0	1	2	2	7								

Similar examples could also be given for load bypassing adds, multiply bypassing loads.

6.7.3.2 Vector Issue Queue (VIQ)

The four-entry vector issue queue (VIQ) handles all AltiVec computational instructions. Two instructions can dispatch to it per cycle, and it can issue two instructions out-of-order per cycle from its bottom two entries (VIQ1–VIQ0) if reservation stations are available. This means an instruction in VIQ1 destined for VIU1 does not have to wait for an instruction in VIQ0 that is stalled behind an instruction waiting for operand availability.

Table 6-21 shows two cases where a vector add and a vector multiply-add (**vmsummbm**) start execution simultaneously (cycles 2 and 3). Note that the load-vector instructions go to the GIQ because its address source operands (**rA** and **rB**) are GPRs. This example also shows the e600 core’s ability to dispatch three instructions with vector targets in a cycle (cycles 0 and 1) as well as to retire three instructions with vector targets (cycle 7).

Table 6-21. VIQ Timing Example

Instruction	0	1	2	3	4	5	6	7
vaddshs V20,V24,V25	D	I	E	F	C			
vmsummbm V10,V11,V12,V13	D	I	E0	E1	E2	E3	C	
lvewx V5,r5,r9	D	I	E0	E1	E2	-	C	
vmsummbm V11,V11,V14,V15	-	D	I	E0	E1	E2	E3	C

Table 6-21. VIQ Timing Example (continued)

Instruction	0	1	2	3	4	5	6	7
vaddshs V21,V26,V27		D	I	E	F	-	-	C
lvevx V5,r6,r9		D	I	E0	E1	E2	-	C

6.7.3.3 Floating-Point Issue Queue (FIQ)

The two-entry floating-point issue queue (FIQ) can accept one dispatched instruction per cycle, and if an FPU reservation station is available, it can also issue one instruction from the bottom FIQ entry.

6.7.4 Completion Unit Resource Requirements

The e600 core completion queue has 16 entries, so as many as 16 instructions can be in the execution window, not counting branches, which execute from the IQ. The following resources are required to avoid stalls in the completion unit; note that the three completion entries are described as CQ0–CQ2, where CQ0 is at the end of the CQ. Requirements for retiring instructions from CQ0–CQ2 are as follows:

- Instruction must have finished execution
- Previous instructions must be retired from the CQ retirement entries.
- Instructions in the CQ must not follow an unresolved predicted branch.
- Instructions in CQ1 and CQ2 must not cause an interrupt.
- Instructions in CQ1 and CQ2 must be integer (IU1 only), floating-point, load, **dcbt**, data streaming, non-folded resolved branches, or AltiVec instructions.
- Number of CR updates from CQ0–CQ2 must not exceed three
- Number of GPR updates from CQ0–CQ2 must not exceed three
- Number of FPR updates from CQ0–CQ2 must not exceed three
- Number of VR updates from CQ0–CQ2 must not exceed three

6.7.4.1 Completion Groupings

The e600 core can retire as many as three instructions per cycle. Only three renames of a given type can be retired per cycle. For example, an **lwzu**, **add**, **subf** sequence has four GPR rename targets, which cannot all retire in the same cycle. The **lwzu** and **add** retire first and **subf** retires one cycle later.

6.7.5 Serialization Effects

The e600 core supports refetch, execution, and store serialization. Store serialization is described in [Section 6.7.6.5.2, “Store Hit Pipeline.”](#)

Refetch serialized instructions include **isync**, **rfi**, **sc**, **mtspr[XER]**, and any instruction that toggles XER[SO]. Refetch serialization forces a pipeline flush when the instruction is the oldest in the machine. Avoid these instructions in performance critical code.

Note that XER[SO] is a sticky bit for XER[OV] updates, so avoiding toggling XER[SO] often means avoiding these instructions (overflow-record, O form).

Execution-serialized instructions wait until the instruction is the oldest in the machine to begin executing. Tables in Section 6.6, “Instruction Latency Summary,” list execution-serialized instructions, which include **mtspr**, **mfspr**, CR logical instructions, and carry consuming instructions (such as **adde**).

Table 6-22 shows the execution of a carry chain. The **addc** executes normally and generates a carry. As an execution-serialized instruction, **adde** must become the oldest instruction (cycle 4) before it can execute (cycle 5). A long chain of carry generation/carry consumption can execute at a rate of one instruction every 3 cycles.

Table 6-22. Serialization Example

Code	0	1	2	3	4	5	6
addc r11,r21,r23	D	I	E	C			
adde r10,r20,r22	D	I	-	-	-	E	C

6.7.6 Execution Unit Considerations

The following sections describes how to optimize use of the execution units.

6.7.6.1 IU1 Considerations

Each of the three IU1s has one reservation station in which instructions are held until operands are available. The IU1s allow a potentially large window for out-of-order execution. IU1 instructions can progress until three IU1 instructions are stuck in the three reservation stations, requiring operands (or until the GIQ or dispatcher stalls for other reasons).

Table 6-23 shows a case where, although two IU1s are blocked, the third makes progress.

Also note that some IU1 instructions take more than one cycle and that some are not fully pipelined. The most common 2-cycle instructions are **sraw** and **srawi**.

The following instructions are not fully pipelined when their record bit is set: **extsb**, **extsh**, **rlwimi**, **rlwinm**, **rlwnm**, **slw**, and **srw**. These instructions return GPR data after the first cycle but continue executing into a second cycle to generate the CR result.

Table 6-23 shows **sraw**, **extsh**, and **extsh**. latency effects. The two **sraw** instructions both take 2 cycles to execute, blocking the **extsh/extsh**. pair from issuing until cycle 3 but allowing the dependent **add** to execute in cycle 3 (See Table 6-5, footnote 3). Note that **extsh**. takes 2 cycles to execute, but that the dependent **subf** can pick up the forwarded GPR value after the first cycle of execution (cycle 4) and execute in cycle 5.

Table 6-23. IU1 Timing Example

Code	0	1	2	3	4	5	6
sraw r1,r20,r21	D	I	E	E	C		
sraw r2,r20,r22	D	I	E	E	C		

Table 6-23. IU1 Timing Example (continued)

Code	0	1	2	3	4	5	6
add r4,r2,r3	D	I	-	E	C		
extsh r5,r25,	F2	D	-	I	E	C	
extsh. r6,r26	F2	D	-	I	E	E	C
subf r7,r5,r6	F2	D	-	I	-	E	C

6.7.6.2 IU2 Considerations

The IU2 has two reservation station entries. Instruction execution is allowed only from the bottom station. Although **mtctr/mtlr** instructions are execution-serialized, if data is available, their values are forwarded to the BPU as soon as they are in the bottom reservation station.

Divides, **mulhwu**, **mulhw**, and **mull** are not fully pipelined; they iterate in execution stage 0 and block other instructions from entering reservation station 0. For example, in [Table 6-20](#), the second multiply issues to IU2 in cycle 2. Because the first multiply still occupies reservation station 0, the second is issued to reservation station 1. When the first multiply enters E1, the second moves down to reservation station 0 and begins execution.

Note that the IU2 takes an extra cycle beyond the latencies listed in [Table 6-5](#) to return CR data and finish. This implies that, as the example in [Section 6.7.3.1, “GPR Issue Queue \(GIQ\),”](#) shows, a 3-cycle instruction such as **mulhw** requires a separate finish stage, even though GPR data is still forwarded and used after 3 execution cycles. In the previous example, instruction 4 executed in cycle 7, the cycle after the dependent instruction 2 went through its third execution stage.

6.7.6.3 FPU Considerations

The FPU has two reservation station entries. Instruction execution is allowed from only the bottom reservation station (reservation station 0).

Like the IU2, the FPU requires a separate finish stage to return CR and FPSCR data, as shown in [Table 6-24](#). However, FPR data produced in E4 (the fifth stage) is ready and can be forwarded directly (if needed) to an instruction entering E0 in the next cycle.

The five-stage scalar FPU pipeline has a 5-cycle latency. However, when the pipeline contains instructions in stages E0–E3, the pipeline stalls and does not allow a new instruction to start in E0 on the following cycle. This bubble limits maximum FPU throughput to four instructions every 5 cycles, as the following code example shows:

```

xxxxxx00      fadd f10,f20,f21
xxxxxx04      fadd f11,f20,f22
xxxxxx08      fadd f12,f20,f23
xxxxxx0c      fadd f13,f20,f24
xxxxxx10      fadd f14,f20,f25
xxxxxx14      fadd f15,f20,f26
xxxxxx18      fadd f16,f20,f27
xxxxxx1c      fadd f17,f20,f28
xxxxxx20      fadd f18,f20,f29
    
```

Table 6-24 shows the timing for this sequence.

Table 6-24. FPU Timing Example

Instruction	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
fadd	D	I	E0	E1	E2	E3	E4	F	C								
fadd	-	D	I	E0	E1	E2	E3	E4	F	C							
fadd	-	-	D	I	E0	E1	E2	E3	E4	F	C						
fadd	-	-	-	D	I	E0	E1	E2	E3	E4	F	C					
fadd	F2	-	-	-	D	I	-	E0	E1	E2	E3	E4	F	C			
fadd	F2	-	-	-	-	D	-	I	E0	E1	E2	E3	E4	F	C		
fadd	F2	-	-	-	-	-	D	-	I	E0	E1	E2	E3	E4	F	C	
fadd	F2	-	-	-	-	-	-	-	D	I	E0	E1	E2	E3	E4	F	C
fadd	F1	F2	-	-	-	-	-	-	-	D	I	-	E0	E1	E2	E3	E4

The FPU is also constrained by the number of FPSCR renames. The e600 core supports four outstanding FPSCR updates. An FPSCR is allocated in the E3 FPU stage and is deallocated at completion. If no FPSCR rename is available, the FPU pipeline stalls. A fully pipelined case such as that in Table 6-24 is not affected, but if something blocks completion it can become a bottleneck. Consider the following code example:

```

xxxxxx00 lfd  f3,0x8(r9)
xxxxxx04 fadd f11,f20,f22
xxxxxx08 fadd f12,f20,f23
xxxxxx0c fadd f13,f20,f24
xxxxxx10 fadd f14,f20,f25
xxxxxx14 fadd f15,f20,f26
xxxxxx18 fadd f16,f20,f27
xxxxxx1c fadd f17,f20,f28
xxxxxx20 fadd f18,f20,f29
    
```

The timing for this sequence in Table 6-25 assumes that the load misses in the data cache. Here, after the first four **fadds**, the e600 core runs out of FPSCR renames and the pipeline stalls. When the load completes, the pipeline restarts after an additional 2-cycle lag.

Table 6-25. FPSCR Rename Timing Example

Instruction	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
lfd	D	I	E0	E1									C			
fadd	D	I	E0	E1	E2	E3	E4	F	-	-	-	-	C			
fadd	-	D	I	E0	E1	E2	E3	E4	F	-	-	-	C			
fadd	-	-	D	I	E0	E1	E2	E3	E4	F	-	-	-	C		
fadd	F2	-	-	D	I	E0	E1	E2	E3	E4	F	-	-	C		
fadd	F2	-	-	-	D	I	-	E0	E1	E2	E3	E4	E4	E4	E4	F
fadd	F2	-	-	-	-	D	-	I	E0	E1	E2	E3	E3	E3	E3	E4
fadd	F2	-	-	-	-	-	D	-	I	E0	E1	E2	E2	E2	E2	E3
fadd	F1	F2	-	-	-	-	-	-	D	I	E0	E1	E1	E1	E1	E2

Note that denormalized numbers can cause problems for the FPU pipeline, so the normal latencies in [Table 6-6](#) may not apply. Output denormalization in the very unlikely worst case can add as many as 4 cycles of latency. Input denormalization takes 4–6 additional cycles, depending on whether 1, 2, or 3 input source operands are denormalized.

6.7.6.4 Vector Unit Considerations

On the e600 core, the four vector execution units are fully independent and fully pipelined. Latencies are given in [Table 6-26](#).

Table 6-26. Vector Execution Latencies

Unit	Latency
VIU1	1
VIU2	4
VFPU	4
VPU	2

VFPU latency is usually 4 cycles, but some instructions, particularly the vector float compares and vector float min/max (see [Table 6-8](#) for a list) have only a 2-cycle latency. This can create competition for the VFPU register forwarding bus. This is solved by forcing a partial stall when a bypass is needed. Consider the following code example:

```

xxxxxx20      vaddfp v10,v11,v12
xxxxxx24      vsubfp v11,v14,v13
xxxxxx28      vaddfp v12,v13,v14
xxxxxx2c      vcmpbfp. v13,v18,v19
xxxxxx30      vmaddfp v14,v20,v21,v14
    
```

[Table 6-27](#) shows the timing for this vector compare bypass/stall situation. In cycle 6 the **vcmp** bypasses from E0 to E3, stalling the **vsubfp** and **vlogfp** for a cycle in stages E1 and E2. Note that an instruction in E1 stalls in E1 under a bypass scenario even if no instruction is in E2.

Table 6-27. Vector Unit Example

Instruction	0	1	2	3	4	5	6	7	8	9	10
vaddfp	D	I	E0	E1	E2	E3	C				
vsubfp	D	-	I	E0	E1	E2	E2	E3	C		
vlogefp	-	D	-	I	E0	E1	E1	E2	E3	C	
vcmpbfp.	-	D	-	-	I	E0	E3	-	-	C	
vmaddfp	F2	-	D	-	-	I	E0	E1	E2	E3	C

6.7.6.5 Load/Store Unit (LSU)

The LSU controls the 32-Kbyte L1 data cache and contains a variety of queues and latches, as shown in [Figure 6-17](#). The core memory subsystem interconnect referred to on the diagram is an interface point to the L2 cache and MPX bus. Note that this is a simplified block diagram, and does not contain the fully detailed LSU microarchitecture.

Instruction execution is allowed only from the lower of the two reservation stations.

Loads that have all required available source operands can start execution. If the load hits in the data cache, data is forwarded to one of the three register rename types. A floating-point load has one additional cycle of latency (4 cycles) beyond that of an integer or vector load (3 cycles). If a stall or hazard occurs, the instruction is typically held in the required address (**rA**) latch.

Loads that miss go to the 5-entry load miss queue (LMQ), where they are held while the line transaction proceeds to the L2 cache and/or the MPX bus. Critical data forwarding can occur from the MPX bus to directly update the required rename. A load that receives critical data can finish.

However (for a cacheable load), the LMQ entry can be deallocated only when the full line returns. When the full line becomes available, the L1 data cache is updated. If an L1 data cache update requires that a line currently in the cache be evicted, that line is cast out and placed into the 6-entry L1 castout queue.

Stores that have the required address source operands (**rA** and possibly **rB**) available start execution similar to loads. However, they are transferred to the 3-entry finished store queue (FSQ). The FSQ holds stores until they have been retired by the completion unit. Once retired, the stores travel through **wb0** and **wb1**, two write-back stages (not shown in [Figure 6-17](#)), while acquiring data (**rS**, **frS**, or **vs**) from the appropriate register file, and are written into the 5-entry committed store queue (CSQ). Stores in the CSQ arbitrate into the L1 data cache. When arbitration is successful, the data is written and the store is removed from the CSQ.

The vector touch engine (VTE) contains the control logic execution of the **dst** instructions.

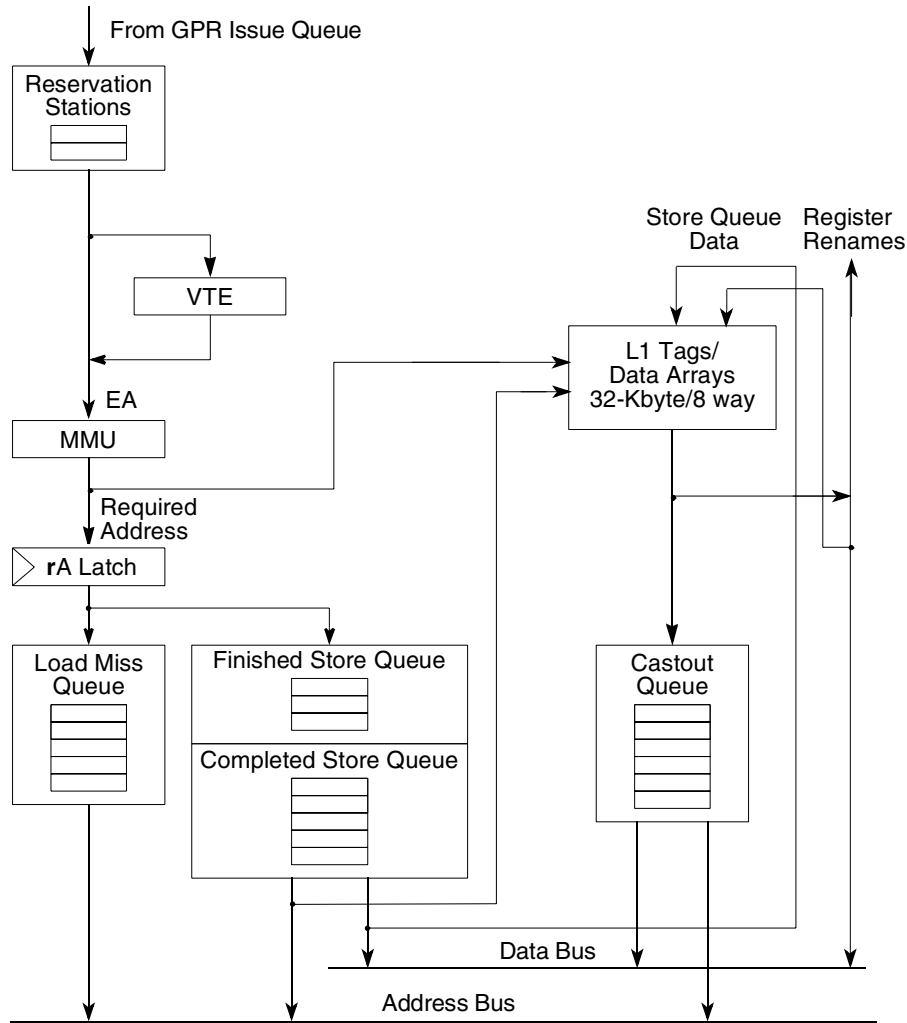


Figure 6-17. LSU Block Diagram

6.7.6.5.1 Load Hit Pipeline

The following code sequence shows the various normal load latencies:

```

xxxxxx00 lfd  f3,0x8(r10)
xxxxxx04 fadd f1,f3,f4
xxxxxx08 lwz  r3,0x4(r11)
xxxxxx0c add  r1,r3,r4
xxxxxx10 subf r5,r11,r6
xxxxxx14 lvev v3,r12,r13
xxxxxx18 vaddsws v1,v3,v4
    
```

As Table 6-28 shows, the load-floating-point latency is 4 and the load-integer and load-vector latency are each 3. Although the load has a 4-cycle latency, it also completes on that fourth cycle. The update has an effective latency of 1. The **lwz** forwards its update target R11 from E0 in cycle 3 to the **subf** instruction, such that it executes in cycle 4.

Table 6-28. Load Hit Pipeline Example

Inst#	Instruction	0	1	2	3	4	5	6	7	8	9	10	11	12	13
0	lfdl	D	I	E0	E1	E2	E3/C								
1	fadd	D	I	-	-	-	-	E0	E1	E2	E3	E4	F	C	
2	lwzu	-	D	I	E0	E1	E2	-	-	-	-	-	-	C	
3	add	-	D	I	-	-	-	E	-	-	-	-	-	C	
4	subf	F2	D	I	-	E	-	-	-	-	-	-	-	-	C
5	lvevx	F2	-	D	I	E0	E1	E2	-	-	-	-	-	-	C
6	vaddsws	F2	-	D	I	-	-	-	E	F	-	-	-	-	C

6.7.6.5.2 Store Hit Pipeline

The pipeline for stores before the data is written to the cache includes several different queues. A store must go through E0 and E1 to handle address generation and translation. It is then placed in the 3-entry FSQ. When the store is the oldest instruction, it can access the store data and update architecture-defined resources (store serialization). From this point on, the store is considered part of architectural state.

However, before the data reaches the data cache, two write-back stages (WB0 and WB1) are needed to acquire the store data and transfer it from the FSQ to the 5-entry CSQ. Arbitration into the data cache from the CSQ is pipelined so a one store per cycle throughput can be maintained. During this arbitration and cache write, stores arbitrate into the data cache from the CSQ and stay there for at least 4 cycles. [Table 6-29](#) shows pipelining of four **stw** instructions to the data cache.

Table 6-29. Store Hit Pipeline Example

Instruction	0	1	2	3	4	5	6	7	8	9	10	11	12	13
stw	D	I	E0	E1	FSQ0/C	WB0	WB1	CSQ0	CSQ0	CSQ0	CSQ0			
stw	-	D	I	E0	E1	FSQ0/C	WB0	WB1	CSQ1	CSQ1	CSQ1	CSQ0		
stw	-	-	D	I	E0	E1	FSQ0/C	WB0	WB1	CSQ2	CSQ2	CSQ1	CSQ0	
stw	-	-	-	D	I	E0	E1	FSQ0/C	WB0	WB1	CSQ3	CSQ2	CSQ1	CSQ0

Floating-point stores are not fully pipelined. The bottleneck is at the FSQ point, where only one floating-point store can be executed every 3 cycles. See [Table 6-30](#) for an example execution of four **stfd** instructions. Vector stores do not suffer from this problem and are fully pipelined (similar to the integer stores as shown in [Table 6-29](#)).

To avoid floating-point store throughput bottlenecks, avoid strings of back-to-back floating-point stores (like that shown in [Table 6-30](#)). Instead, intermix floating-point stores with other instructions wherever possible. For maximum store throughput, use vector stores.

Table 6-30. Execution of Four stfd Instructions

Inst#	Instruction	Cycle Number									
		0	1	2	3	4	5	6	7	8	9
0	stfd	D	I	E0	E1	FSQ0/C	WB0	WB1	CSQ0	CSQ0	CSQ0
1	stfd	-	D	I	E0	E1	FSQ0	FSQ0	FSQ0/C	WB0	WB1
2	stfd	-	-	D	I	E0	E1	FSQ1	FSQ1	FSQ0	FSQ0
3	stfd	-	-	-	D	I	E0	E1	FSQ2	FSQ1	FSQ1
		10	11	12	13	14	15	16	17	18	19
0	stfd	CSQ0									
1	stfd	CSQ1	CSQ0	CSQ0	CSQ0						
2	stfd	FSQ0/C	WB0	WB1	CSQ1	CSQ0	CSQ0	CSQ0			
3	stfd	FSQ1	FSQ0	FSQ0	FSQ0/C	WB0	WB1	CSQ1	CSQ0	CSQ0	CSQ0

6.7.6.5.3 Load/Store Interaction

When loads and stores are intermixed, the stores normally lose arbitration to the cache. A store that repeatedly loses arbitration can stay in the CSQ much longer than 4 cycles, which is not normally a performance problem because a store in the CSQ is effectively part of the architecture-defined state. However, sometimes—including if the CSQ fills up or if a store causes a pipeline stall (as in a partial address alias case of store to load)—the arbiter gives higher priority to the store, guaranteeing forward progress.

Also, accesses to the data cache are pipelined (two stages) such that back-to-back loads and back-to-back stores are fully pipelined (single-cycle throughput). However, a store followed by a load cannot be performed in subsequent clock cycles. Loads have higher priority than stores and the LSU store queues stage store operations until a cache cycle is available. When the LSU store queues become full, stores take priority over subsequent loads.

From an architectural perspective, when a load address aliases to a store address the load needs to read the store data rather than the data in the cache. A store can forward only after acquiring its data, which means forwarding happens only from the CSQ. Additionally, the load address and size must be contained within the store address and size for store forwarding to occur. If the alias is only a partial alias (for example an **stb** and an **lwz**) the load stalls. [Table 6-31](#) shows a forwardable load/store alias, where the load stalls in E1 for 3 cycles until the store arrives in CSQ0 and can forward its data.

Table 6-31. Load/Store Interaction (Assuming Full Alias)

Instruction	0	1	2	3	4	5	6	7	8
stw r3,0x0(r9)	E0	E1	FSQ0/C	WB0	WB1	CSQ0	CSQ0	CSQ0	CSQ0
lwz r4,0x0(r9)	I	E0	E1	E1	E1	E1	E2	C	

6.7.6.5.4 Misalignment Effects

Misalignment, particularly the back-to-back misalignment of loads, can cause strange performance effects. The e600 core splits misaligned transactions into two transactions, so misaligned load latency is at least 1 cycle greater than the default latency. On the e600 core, misalignment typically occurs when an access crosses a double-word boundary. [Table 6-32](#) shows what is considered misaligned based on the EA of the access. Note that vector transactions ignore non-size-aligned low-order address bits and so are not considered misaligned.

Table 6-32. Misaligned Load/Store Detection

EA[29:31]	Byte	Half Word	Word	Double Word	Quad-Word/Bus
000	—	—	—	—	—
001	—	—	Multi/floating-point exception	Floating-point exception	Align to quad word
010	—	—	Multi/floating-point exception	Floating-point exception	Align to quad word
011	—	—	Multi/floating-point exception	Floating-point exception	Align to quad word
100	—	—	—	Misaligned	Align to quad word
101	—	—	Misaligned or multi/floating-point exception	Floating-point exception	Align to quad word
110	—	—	Misaligned or multi/floating-point exception	Floating-point exception	Align to quad word
111	—	Misaligned	Misaligned or multi/floating-point exception	Floating-point exception	Align to quad word

Future generations of high-performance microprocessors that implement the PowerPC architecture may experience greater misalignment penalties.

6.7.6.5.5 Load Miss Pipeline

The e600 core supports as many as five outstanding load misses in the load miss queue (LMQ). [Table 6-33](#) shows a load followed by a dependent **add**. Here, the load misses in the data cache and the full line are reloaded from the L2 cache back into the data cache. The load L2 cache hit latency is effectively 11 cycles if ECC is disabled, and 12 cycles if ECC is enabled. The instruction fetch latency for an L2 hit is 15/16 cycles.

Table 6-33. Data Cache Miss, L2 Cache Hit Timing

Instruction	0	1	2	3–9/10	10/11	11/12	12/13
lwz r4,0x0(r9)	E0	E1	Miss	LMQ0	LMQ0/E2	C	
add r5,r4,r3	—	—	—	—	—	E	C

If a load misses in the L1 data cache and in the L2 data cache, critical data forwarding occurs, followed shortly by the rest of the line. The example in [Table 6-34](#) shows the load being fetched from main memory, where N is the arbitrarily long time it takes to return from the MPX bus.

Table 6-34. Data Cache Miss, L2 Cache Miss, Main Memory Fetch Timing

Instruction	0	1	2	3–N	N+1	N+2	N+3	N+4–N+5
lwz r4,0x0(r9)	E0	E1	Miss	LMQ0	LMQ0/E2	LMQ0/C	LMQ0	LMQ0
add r5,r4,r3	—	—	—	—	—	E	C	—

Note that the LMQ0 entry for the load remained allocated for 4 cycles after the critical data arrived in cycle 32. This is because with a 4:1 bus:core ratio, there is a 4-cycle gap between critical data and full line data, and the LMQ entry is only deallocated when the full line has returned.

If a load/store miss aliases to the same line as a previously outstanding miss, the LSU halts new access until this stall condition is resolved. The example in [Table 6-35](#) contains a series of loads, where the data starts in main memory.

Table 6-35. Load Miss Line Alias Example

Inst#	Instruction	Cycle Number							
		0	1	2	3–N	N+1	N+2	N+3	N+4–N+5
0	lwz r3,0x0(r9)	E0	E1	Miss	LMQ0	LMQ0/E2	LMQ0/C	LMQ0	LMQ0
1	add r4,r3,r20						E	C	
2	lwz r5,0x4(r9)	I	E0	E1	E1	E1	E1	E1	E1
3	add r6,r5,r4	I							
4	lwz r7,0x20(r9)	D	I	E0	E0	E0	E0	E0	E0
5	add r8,r7,r6	D	I						
		N+6–N+8	N+9	N+10	N+11	N+12–N+30	N+31	N+32	N+33
0	lwz r3,0x0(r9)								
1	add r4,r3,r20								
2	lwz r5,0x4(r9)	E1	E2	C					
3	add r6,r5,r4			E	C				
4	lwz r7,0x20(r9)	E0	E1	Miss	LMQ0	LMQ0	LMQ0/E2	LMQ0/C	LMQ0
5	add r8,r7,r6							E	C

Note that instruction 2 stalls in stage E1 (in the rA latch in [Table 6-35](#)). This stall is due to the fact that the line miss caused by instruction 0 is the same line that instruction 2 requires. Instruction 2 does not finish execution until cycle N+9—8 cycles after instruction 0. This delay is due to 2 major components. The first delay component is that instruction 0 finished by using critical forwarded data, whereas instruction 2 must wait for the full cache line to appear before it can start execution (a 4-cycle delay, in this example). The second delay component is also due to the cache being updated and a pipeline restart.

The second problem that this example shows is that the misses are not fully pipelined. Instructions 0 and 4 miss in the data cache and L2 cache but are fetched from main memory. The stall caused by the line miss alias between instructions 0 and 2 has caused the miss for instruction 4 to delay its access start by

many cycles. A simple reordering of the code, as the example in [Table 6-36](#) allows the two load misses to pipeline to main memory, improving performance by nearly 50%.

Table 6-36. Load Miss Line Alias Example With Reordered Code

Inst#	Instruction	Cycle Number						
		0	1	2	3	4–N	N+1	N+2
0	lwz r3,0x0(r9)	E0	E1	Miss	LMQ0	LMQ0	LMQ0/E2	LMQ0/C
1	add r4,r3,r20							E
2	lwz r7,0x20(r9)	I	E0	E1	Miss	LMQ1	LMQ1	LMQ1
3	lwz r5,0x4(r9)	D	I	E0	E1	E1	E1	E1
4	add r6,r5,r4	D	I					
5	add r8,r7,r6	D	I					
		N+3	N+4–N+5	N+6–N+8	N+9	N+10	N+11	N+12
0	lwz r3,0x0(r9)	LMQ0	LMQ0					
1	add r4,r3,r20	C						
2	lwz r7,0x20(r9)	LMQ1	LMQ1	LMQ1	LMQ1	LMQ1/E2	LMQ1/C	LMQ1
3	lwz r5,0x4(r9)	E1	E1	E1	E2		C	
4	add r6,r5,r4					E	C	
5	add r8,r7,r6						E	C

This type of stall is common in some specific kinds of code, including simple data streaming or striding array accesses. For example, a long stream of vector loads with addresses incrementing by 16 bytes (a quad word) per load stalls every other load stalled in this manner, and no miss pipelining occurs. This stall causes an even larger performance bottleneck when cache misses are required to go to the MPX bus and when missed opportunities to pipeline MPX bus misses occur. This performance problem can be solved by code reordering as shown in [Table 6-36](#) or by the use of prefetch instructions (**dcbt** or **dst**).

The e600 core does back-end allocation of the L1 data cache, which means that it selects the line replacement (and pushes to the six-entry castout queue as needed) only when a reload returns. Because any new miss transaction may later require a castout, a new miss is not released to the core memory subsystem until a castout queue slot is guaranteed.

6.7.6.5.6 Store Miss Pipeline

The e600 core supports two outstanding store misses, which use committed store queue entries 0 and 1 (CSQ0 and CSQ1). For applications needing considerable store-miss bandwidth to a cacheable memory region and doing read/modify/write operations, consider using **dst** or **dcbtst** instructions to prefetch needed lines. This allows the use of the 5-entry LMQ to provide miss pipelining.

However, using the **dcbz** instruction is strongly encouraged for storing to a new cache block, when the entire block will be written (and not be read before being written). The **dcbz** creates an address-only transaction that avoids waiting for data to be read from the L2 or the MPX bus and then updating the data

cache only to be immediately overwritten by the store. Using the **dstst** instruction is discouraged (either the operation is read/modify/write, in which case **dbt** or **dst** instructions should be used; or the operation is write-only, in which case prefetching the data is a bad idea and a **dcbz** should instead be used).

Table 6-37 shows a series of cacheable stores, where the stores miss in the data cache and L2 cache and take an arbitrarily long time (N cycles) to return from the MPX bus. Instructions 0–7 are all storing data to the same cache line, while instructions 8–9 are storing data to the next adjacent cache line. Instruction 0 data cache access occurs in cycle 7 and the miss is transferred to the lower levels of the cache hierarchy starting in cycle 9.

Table 6-37. Store Miss Pipeline Example

Inst#	Instruction	Cycle Number							
		0	1	2	3	4	5	6	
0	stw r10,0x0(r8)	E0	E1	FSQ0/C	WB0	WB1	CSQ0	CSQ0	
1	stwu r11,0x4(r8)	I	E0	E1	FSQ0/C	WB0	WB1	CSQ1	
2	stwu r12,0x4(r8)	D	I	E0	E1	FSQ0/C	WB0	WB1	
3	stwu r13,0x4(r8)		D	I	E0	E1	FSQ0/C	WB0	
4	stwu r14,0x4(r8)			D	I	E0	E1	FSQ0/C	
5	stwu r15,0x4(r8)				D	I	E0	E1	
6	stwu r16,0x4(r8)					D	I	E0	
7	stwu r17,0x4(r8)						D	I	
8	stwu r18,0x4(r8)							D	
9	stwu r19,0x4(r8)								
		7	8	9	10	11	12	13	
0	stw r10,0x0(r8)	CSQ0	CSQ0	CSQ0	CSQ0	CSQ0	CSQ0	CSQ0	
1	stwu r11,0x4(r8)	CSQ1	CSQ1	CSQ1	CSQ1	CSQ1	CSQ1	CSQ1	
2	stwu r12,0x4(r8)	CSQ2	CSQ2	CSQ2	CSQ2	CSQ2	CSQ1	CSQ1	
3	stwu r13,0x4(r8)	WB1	CSQ3	CSQ3	CSQ3	CSQ3	CSQ2	CSQ2	
4	stwu r14,0x4(r8)	WB0	WB1	CSQ4	CSQ4	CSQ4	CSQ3	CSQ3	
5	stwu r15,0x4(r8)	FSQ0/C	WB0	WB1	WB1	WB1	CSQ4	CSQ4	
6	stwu r16,0x4(r8)	E1	FSQ0	FSQ0	FSQ0	FSQ0	FSQ0/C	WB0	
7	stwu r17,0x4(r8)	E0	E1	FSQ1	FSQ1	FSQ1	FSQ1	FSQ0	
8	stwu r18,0x4(r8)	I	E0	E1	FSQ2	FSQ2	FSQ2	FSQ1	
9	stwu r19,0x4(r8)	D	I	E0	E1	E1	E1	E1	
		14	15	16	17	18	19	20	
0	stw r10,0x0(r8)	CSQ0	CSQ0	CSQ0	CSQ0	CSQ0	CSQ0	CSQ0	
1	stwu r11,0x4(r8)	CSQ1	CSQ1	CSQ1	CSQ1	CSQ1	CSQ1	CSQ1	
2	stwu r12,0x4(r8)	CSQ1	CSQ1	CSQ1	CSQ1	CSQ1	CSQ1	CSQ1	

Table 6-37. Store Miss Pipeline Example (continued)

Inst#	Instruction	Cycle Number						
3	stwu r13,0x4(r8)	CSQ2	CSQ2	CSQ1	CSQ1	CSQ1	CSQ1	CSQ1
4	stwu r14,0x4(r8)	CSQ2	CSQ2	CSQ1	CSQ1	CSQ1	CSQ1	CSQ1
5	stwu r15,0x4(r8)	CSQ3	CSQ3	CSQ2	CSQ2	CSQ2	CSQ2	CSQ1
6	stwu r16,0x4(r8)	WB1	CSQ4	CSQ3	CSQ3	CSQ2	CSQ2	CSQ1
7	stwu r17,0x4(r8)	FSQ0/C	WB0	WB1	CSQ4	CSQ3	CSQ3	CSQ2
8	stwu r18,0x4(r8)	FSQ1	FSQ0	FSQ0/C	WB0	WB1	CSQ4	CSQ3
9	stwu r19,0x4(r8)	FSQ2	FSQ1	FSQ1	FSQ0	FSQ0/C	WB0	WB1
		21	22	23–24	25	26–N	N+1–N+4	N+5
0	stwu r10,0x0(r8)	CSQ0	CSQ0	CSQ0	CSQ0	CSQ0		
1	stwu r11,0x4(r8)	CSQ1	CSQ1	CSQ1	CSQ1	CSQ1	CSQ0	
2	stwu r12,0x4(r8)	CSQ1	CSQ1	CSQ1	CSQ1	CSQ1	CSQ0	
3	stwu r13,0x4(r8)	CSQ1	CSQ1	CSQ1	CSQ1	CSQ1	CSQ0	
4	stwu r14,0x4(r8)	CSQ1	CSQ1	CSQ1	CSQ1	CSQ1	CSQ0	
5	stwu r15,0x4(r8)	CSQ1	CSQ1	CSQ1	CSQ1	CSQ1	CSQ0	
6	stwu r16,0x4(r8)	CSQ1	CSQ1	CSQ1	CSQ1	CSQ1	CSQ0	
7	stwu r17,0x4(r8)	CSQ2	CSQ1	CSQ1	CSQ1	CSQ1	CSQ0	
8	stwu r18,0x4(r8)	CSQ3	CSQ2	CSQ2	CSQ2	CSQ2	CSQ1	CSQ0
9	stwu r19,0x4(r8)	CSQ4	CSQ3	CSQ3	CSQ2	CSQ2	CSQ1	CSQ0

In this example a store gathering opportunity occurs as a store miss has happened. See [Section 6.4.4.2, “Store Gathering,”](#) and [Section 3.1.2.3, “Store Gathering/Merging,”](#) for rules about how and when a store gathering opportunity can be taken advantage of. To maximize the potential for store gathering, stores to adjacent datum should not be interleaved with other stores.

The first store gathering occurs in cycle 12, where adjacent store instructions 1 and 2 are gathered to the same CSQ entry (1). The gathering also occurs in cycles 14, 16, 18, 20, and 22. In cycle 22, an entire cache line has gathered except for the first access that caused the miss. The second cache line access (for instructions 8 and 9) gathers in cycle 25. Given a sufficiently long miss latency for instruction 0, a full line could have been gathered into entry CSQ2, if instructions 10–15 had been shown and also were storing to the same line as instructions 8–9.

As the data reload occurs in cycle N, instruction 0 is removed from the CSQ. It takes another 4 cycles for the next store access (the gathered access of instructions 1–7) to restart and write into the cache.

6.7.6.5.7 DST Instructions and the Vector Touch Engine (VTE)

The e600 core VTE engine can only initiate an access every 3 cycles. However, due to miss-handling differences described in [Section 6.7.6.5.5, “Load Miss Pipeline,”](#) the engine may fall behind and conflict

with the work of the core. Therefore, retuning the **dst** may be necessary to optimize the performance of the e600 core.

Also, note the information on hardware prefetching in [Section 6.7.7.2, “Hardware Prefetching.”](#) Although hardware prefetching is useful for many general-purpose applications, it may not be the best choice for when active prefetch control through software is attempted. Hardware prefetching can sometimes interfere with the attempt by the **dst** engine to keep the MPX bus busy with specific prefetch transactions, especially for **dst** strides larger than one cache block or transient **dst** operations. Experimentation is encouraged, but in this instance the best solution may be to disable hardware prefetching.

6.7.7 Core Memory Subsystem Considerations

The two-level cache implementation affects instruction fetching, the loading and storing source, and destination operands, as described in the following sections.

6.7.7.1 L2 Cache Effects

For the e600 core, the unified 1-Mbyte on-board L2 cache has 8-way set associativity and 64-byte lines (with two sectors/line). This implies 16384 lines (1 M/64) and 2048 sets (1 M/64/8). [Figure 3-17](#) shows the L2 cache organization. Each line has two sectors with one tag per line but separate valid and dirty bits for each sector. Because of the sectoring, code uses more of the L2 storage if spatial locality is characterized by use of the adjacent 32-byte line.

A load that misses the L1 but hits the L2 causes a full line reload. Its latency is ideally 11 cycles with ECC disabled and 12 cycles with ECC enabled assuming higher priority L2 traffic. See [Table 6-33](#).

An access missing the L2 goes to the MPX bus to fetch the needed 32-byte sector.

The L2 cache uses a pseudo-random replacement algorithm. With 8-way set associativity, a miss randomly replaces 1 of 8 ways. This works well for smaller working set sizes, but for working set sizes close to the size of the cache, the hit rate is not quite as good. Imagine a 64-Kbyte array structure and a byte-striding access pattern that loops over the array several times. The access of the first 32 Kbytes (256 Kbyte/8 ways) misses and loads correctly, but the second 32 Kbytes has a 1 in 8 chance per set of knocking out an index of the first 32 Kbytes. This means that the first pass is likely to leave 93.75% of the 64-Kbyte structure in the L2 cache; a second pass is likely to leave 99.8% of the structure in the L2 cache.

For a 128-Kbyte object, 82.8% is left in the L2 cache after one pass, but a 256-Kbyte object only slightly less than 2/3 of the structure is left in the L2 cache. However, in both cases the percentages improve with subsequent passes.

6.7.7.2 Hardware Prefetching

The e600 core supports alternate sector prefetching from the L2 cache. Because the L2 cache is two-sectored, an access requesting a 32-byte line from the L1 that also misses in the L2 can generate a prefetch (if enabled) for the alternate sector as needed. As many as three outstanding prefetches are allowed.

The example in [Table 6-38](#) shows timing when the loads miss both levels of the cache hierarchy and go to the MPX bus. Hardware prefetching is disabled. The load misses to the MPX bus are serialized by the load miss line alias stall (instruction 2 on instruction 0).

Table 6-38. Timing for Load Miss Line Alias Example

Inst#	Instruction	Cycle Number							
		0	1	2	3–81	82	83	84	85–99
0	lwz r3,0x0(r9)	E0	E1	Miss	LMQ0	LMQ0/E2	LMQ0/C	LMQ0	LMQ0
1	add r4,r3,r20						E	C	
2	lwz r5,0x4(r9)	I	E0	E1	E1	E1	E1	E1	E1
3	add r6,r5,r4	I							
4	lwz r7,0x20(r9)	D	I	E0	E0	E0	E0	E0	E0
5	add r8,r7,r6	D	I						
		100–102	103	104	105	106–184	185	186	187
0	lwz r3,0x0(r9)								
1	add r4,r3,r20								
2	lwz r5,0x4(r9)	E1	E2	C					
3	add r6,r5,r4			E	C				
4	lwz r7,0x20(r9)	E0	E1	Miss	LMQ0	LMQ0	LMQ0/E2	LMQ0/C	LMQ0
5	add r8,r7,r6							E	C

However, if hardware prefetching is enabled, hardware starts prefetching the line desired by instruction 4 before instruction 4 accesses (and misses) in the L1 data cache, thus parallelizing some serialized bus accesses. In the example shown in [Table 6-39](#), with prefetching enabled, performance improves by about 40%. In this case, the prefetch was not finished when instruction 4 went to the L2 cache, so the load is forced to stall while the prefetch bus access completes. However, in other cases, the hardware prefetch is entirely finished, allowing subsequent loads to have the access time of an L2 cache hit. In general, hardware prefetch benefits are very dependent on what type of applications are run and how the system is configured.

Table 6-39. Hardware Prefetching Enable Example

Inst#	Instruction	Cycle Number							
		0	1	2	3–81	82	83	84	85–99
0	lwz r3,0x0(r9)	E0	E1	Miss	LMQ0	LMQ0	LMQ0/E2	LMQ0/C	LMQ0
1	add r4,r3,r20							E	C
2	lwz r5,0x4(r9)	I	E0	E1	E1	E1	E1	E1	E1
3	add r6,r5,r4	I							
4	lwz r7,0x20(r9)	D	I	E0	E0	E0	E0	E0	E0

Table 6-39. Hardware Prefetching Enable Example (continued)

Inst#	Instruction	Cycle Number							
5	add r8,r7,r6	D	I						
		100–102	103	104	105	106–133	134	135	136
0	lwz r3,0x0(r9)								
1	add r4,r3,r20								
2	lwz r5,0x4(r9)	E1	E2	C					
3	add r6,r5,r4			E	C				
4	lwz r7,0x20(r9)	E0	E1	Miss	LMQ0	LMQ0	LMQ0/E2	LMQ0/C	LMQ0
5	add r8,r7,r6							E	C

Hardware prefetching is often preferable. However, sometimes an unnecessary prefetch transaction can delay a later-arriving demand transaction and slow down the processor. Also, as described in [Section 6.7.6.5.7, “DST Instructions and the Vector Touch Engine \(VTE\),”](#) if software prefetching is used, hardware prefetching may sometimes provide more interference than benefit.

Chapter 7

AltiVec Technology Implementation

The AltiVec technology, a short vector parallel architecture, extends the instruction set architecture (ISA) of the PowerPC architecture. The AltiVec ISA is based on separate vector/SIMD-style (single instruction stream, multiple data streams) execution units that have high-data parallelism. That is, the AltiVec technology operations can perform on multiple data elements in a single instruction. The term ‘vector’ in this document refers to the spatial parallel processing of short, fixed-length, one-dimensional matrices performed by an execution unit. It should not be confused with the temporal parallel (pipelined) processing of long, variable-length vectors performed by classical vector machines. High degrees of parallelism are achievable with simple, in-order instruction dispatch and low instruction bandwidth. However, the ISA is designed to not impede additional parallelism through superscalar dispatch in multiple execution units or multithreaded execution unit pipelines.

The e600 core AltiVec ISA supports the out-of-order issue of AltiVec instructions. Instructions can be issued out of order from the bottom two VIQ entries (VIQ1–VIQ0). An instruction in VIQ1 destined for VIU1 does not have to wait for an instruction in VIQ0 that is stalled behind an instruction waiting for operand availability.

The AltiVec specification is defined in the *AltiVec Technology Programming Environments Manual*. That document describes but does not require many aspects of a preferred implementation. The e600 core implements the following key features of preferred implementation:

- All data paths and execution units are 128 bits wide.
- There are four independent AltiVec subunits for executing AltiVec instructions: permute, complex, simple, and float.
- The core memory subsystem is redesigned to provide high bandwidth.
- The data stream touch instructions, **dst(t)** (for loads) and **dstst(t)** (for stores) are implemented in their full, four-tag form.

The AltiVec instruction set both defines entirely new resources and extends the functionality of the PowerPC architecture. These changes are described in the following sections.

7.1 AltiVec Technology and the Programming Model

The following sections describe how the AltiVec technology affects features of the programming model as described in [Chapter 2, “Registers.”](#) Although the AltiVec specification describes four optional user-mode SPRs for thread management, the e600 core does not implement these registers.

7.1.1 Register Set

The incorporation of AltiVec technology affects the register set in the e600 core as described in the following sections. These features are detailed in the *AltiVec Programming Environments Manual*.

7.1.1.1 Changes to the Condition Register

AltiVec vector-compare operations with Rc set can update condition register field 6 (CR[6]) in user mode.

7.1.1.2 Addition to the Machine State Register

The AltiVec available bit, MSR[VEC], indicates the availability of the AltiVec instruction set. Its default state for the e600 core is a zero (not available). It can be set by the supervisor-level **mtmsr** instruction.

7.1.1.3 Vector Registers (VRs)

The AltiVec programming model defines vector registers (VRs) that are used as source and destination operands for AltiVec load, store, and computational instructions.

Figure 7-1 shows the 32 registers of the vector register file (VRF). Each is 128 bits wide and can hold sixteen 8-bit elements, eight 16-bit elements, or four 32-bit elements.

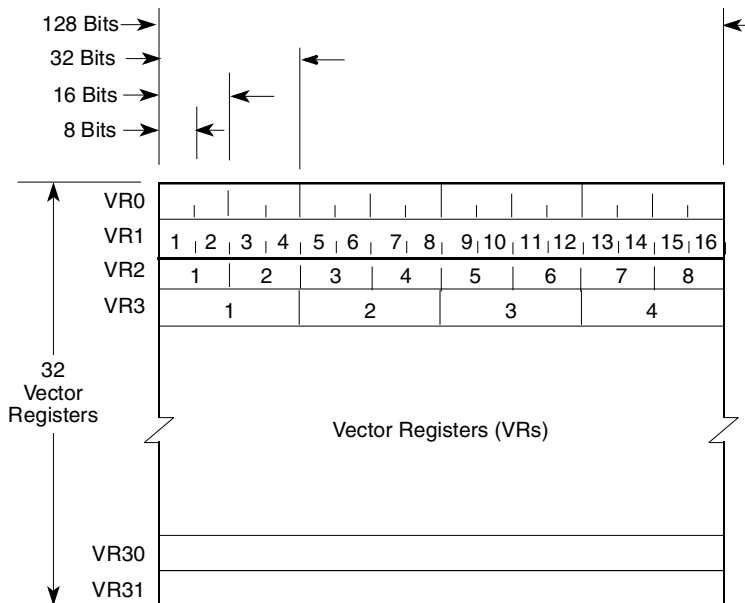


Figure 7-1. Vector Registers (VRs)

7.1.1.4 Vector Status and Control Register (VSCR)

The vector status and control register (VSCR) is a 32-bit vector register (not an SPR) that functions similarly to the FPSCR and is accessed by AltiVec instructions. The Move from Vector Status and Control Register (**mfvscr**) and Move to Vector Status and Control Register (**mtvscr**) instructions are provided to move the contents of the VSCR from and to the least-significant bits of a vector register. The VSCR is shown in Figure 7-2.

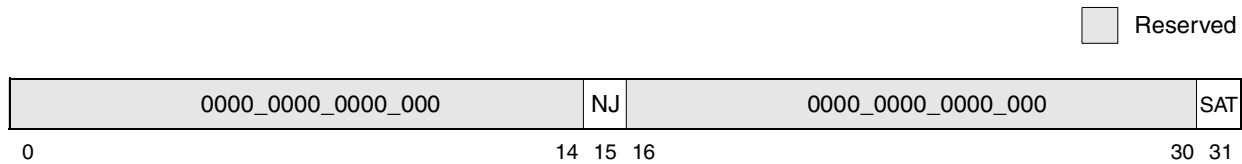


Figure 7-2. Vector Status and Control Register (VSCR)

The VSCR has two defined bits, the AltiVec non-Java mode bit (VSCR[NJ]) and the AltiVec saturation bit (VSCR[SAT]). The remaining bits are reserved.

VSCR bits are described in Table 7-1.

Table 7-1. VSCR Field Descriptions

Bits	Name	Description
0–14	—	Reserved. The handling of reserved bits is the same as that for other PowerPC registers. Software is permitted to write any value to such a bit. A subsequent reading of the bit returns 0 if the value last written to the bit was 0 and returns an undefined value (0 or 1) otherwise.
15	NJ	Non-Java. This bit determines whether AltiVec floating-point operations are performed in a Java-compliant mode or a possibly faster non-Java mode. 0 Java-compliant mode (default). In this mode, the AltiVec assist interrupt is enabled. The AltiVec assist interrupt allows software to handle denormalized values as specified in the Java standard. 1 Non-Java mode. If an element in a source vector register contains a denormalized value, the value 0 is used instead. If an instruction causes an underflow condition, the corresponding element in the target VR is cleared to 0. In both cases the 0 has the same sign as the denormalized or underflowing value.
16–30	—	Reserved. The handling of reserved bits is the same as that for other PowerPC registers. Software is permitted to write any value to such a bit. A subsequent reading of the bit returns 0 if the value last written to the bit was 0 and returns an undefined value (0 or 1) otherwise.
31	SAT	Saturation. This sticky status bit indicates that a field in a saturating instruction saturated since the last time SAT was cleared. It is sticky in that when SAT = 1, it remains set to 1 until it is cleared to 0 by an mtvscr instruction. 0 Indicates no saturation occurred; mtvscr can explicitly clear this bit. 1 The AltiVec saturate instruction is set when saturation occurs for the results of one of the AltiVec instructions having 'saturation' in its name, as follows: Move To VSCR (mtvscr) Vector Add Integer with Saturation (vaddubs , vadduhs , vadduws , vaddsbs , vaddshs , vaddsws) Vector Subtract Integer with Saturation (vsububs , vsubuhs , vsubuws , vsubsbs , vsubshs , vsubsws) Vector Multiply-Add Integer with Saturation (vmhaddshs , vmhraddshs) Vector Multiply-Sum with Saturation (vmsumuhs , vmsumshs , vmsumsws) Vector Sum-Across with Saturation (vsumswws , vsum2sws , vsum4sbs , vsum4shs , vsum4ubs) Vector Pack with Saturation (vpkuhs , vpkuws , vpkshs , vpksws , vpkshs , vpksws) Vector Convert to Fixed-Point with Saturation (vctuxs , vctxs)

7.1.1.5 Vector Save/Restore Register (VRSAVE)

The vector save/restore register (VRSAVE) is a user-mode register used to assist application and operating system software in saving and restoring the architectural state across process context-switched events. Shown in Figure 7-3, VRSAVE is a 32-bit special-purpose register (SPR 256) entirely maintained and managed by software.

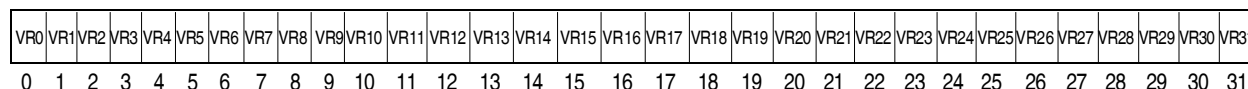


Figure 7-3. Vector Save/Restore Register (VRSAVE)

VRSAVE bit settings are shown in Table 7-2.

Table 7-2. VRSAVE Bit Settings

Bits	Name	Description
0–31	VR	Determine which VRs are used in the current process. 0 Not being used for the current process 1 Used for the current process

7.1.2 AltiVec Instruction Set

The e600 core implements all of the defined AltiVec instructions. The AltiVec instruction set has no optional instructions; however, a few instructions associated with the load/store model are defined to allow significant differences between implementations. The following sections describe the e600 core’s implementation of these options.

AltiVec instructions are primarily user-level and are divided into the following categories:

- Vector integer arithmetic instructions—These include arithmetic, logical, compare, rotate, and shift instructions.
- Vector floating-point arithmetic instructions
- Vector load and store instructions
- Vector permutation and formatting instructions—These include pack, unpack, merge, splat, permute, select, and shift instructions.
- Processor control instructions—These instructions are used to read and write from the VSCR.
- Memory control instructions—These instructions are used for managing caches (user- and supervisor-level).

7.1.2.1 LRU Instructions

The AltiVec architecture suggests that the **lvxl** and **stvxl** instructions differ from other AltiVec load and store instructions in that they leave data cache entries in a least recently used (LRU) state instead of a most recently used state (MRU). This is used to identify data known to have little reuse and poor caching characteristics.

On the e600 core, these instructions follow the cache allocation and replacement policies described in [Section 3.5, “L1 Cache Operation,”](#) but they leave their addressed data cache entries in the LRU state. In addition, all LRU instructions are also interpreted to be transient and are treated as described in [Section 7.1.2.2, “Transient Instructions and Caches.”](#)

7.1.2.2 Transient Instructions and Caches

The e600 core supports both static and transient memory access behavior as defined by the AltiVec technology.

A static memory access assumes a reasonable degree of locality and that the data will be needed several times over a relatively long period. A transient memory reference has poor locality and is likely to be referenced few times or over a short period of time.

For transient memory accesses that miss in the L1 cache, the e600 core allocates (and loads) the line in the L1 cache and marks it as LRU. The core does not allocate entries in the L2 cache for transient accesses that miss. If the L1 cache line is modified and the line is the next candidate for replacement, a castout occurs to system memory. Note that the e600 core writes back to memory in this case and does not allocate the L1 castout in the L2 cache.

The following instructions are interpreted to be transient:

- **lvxl** and **stvxl**
- **dstt** and **dststt** (transient forms of the two-data stream touch instructions). These are described in detail in the following section.

The AltiVec architecture specifies the data stream touch instructions **dst(t)** and **dstst(t)** and two data stream stop (**dss(all)**) instructions. The e600 core implements all of them. The term **dstx** used below refers to all of the data stream touch instructions. The T field in the **dstx** instruction is used as the transient hint bit indicator.

The instructions summarized in this section provide user-level programs with the ability to manage caches within the core; see Chapter 5, “Cache Model and Memory Coherency,” in *The Programming Environments Manual* for more information about cache topics.

Bandwidth between the processor and memory is managed explicitly through the use of cache management instructions that provide a way to indicate to the cache hardware how it should prefetch and prioritize the writeback of data. The principal instruction for this purpose is the software-directed cache prefetch Data Stream Touch (**dst**). Other related instructions are provided for complete control of the software-directed cache prefetch mechanism.

Table 7-3 summarizes the directed prefetch cache instructions defined by the AltiVec VEA. Note that these instructions are accessible to user-level programs.

Table 7-3. AltiVec User-Level Cache Instructions

Name	Mnemonic	Syntax	Implementation Notes
Data Stream Touch (non-transient)	dst	rA,rB,STRM	—
Data Stream Touch (transient)	dstt	rA,rB,STRM	Used for last access
Data Stream Touch for Store (non-transient)	dstst	rA,rB,STRM	Not recommended for use in the e600 core
Data Stream Touch for Store (transient)	dststt	rA,rB,STRM	Not recommended for use in the e600 core
Data Stream Stop (one stream)	dss	STRM	—
Data Stream Stop (all streams)	dssall	STRM	—

7.1.2.3 Data Stream Touch Instructions

Note that, in general, prefetching data to which the program is performing only store instructions does not help and can sometimes hinder performance. User-level programs should not use the touch-for-store prefetches (**dstt**, **dstst**, and **dststt**) unless the program is performing loads and stores to the data that is being prefetched. If the user is performing only stores to the data, performance is almost certainly better if the data is not prefetched and the stores are performed independently. In this case, a **dcbz** instruction is often the best method to initialize the cache block without creating an external memory access request.

So, in general, touch-for-store instructions (**dstt**, **dstst**, and **dststt**) should not be used. The only interrupt is when prefetching data that is going to be both loaded from and then stored to. Otherwise, a programmer should use the normal touch-for-load instruction (**dst**) to prefetch data that the program is loading.

If `HID0[NOPDST] = 1`, all subsequent **dstx** instructions are treated as no-ops, and all previously executed **dst** streams are canceled. This no-op means that the touch does not cause a load operation and cannot perform address translation. Therefore, no table search operations are initiated, and no page table entry (PTE) referenced bits are set.

The **dstx** instructions are broken into one or more self-initiated **dcbt**-like touch line fetches by the core memory subsystem. When the **dstx** instruction is dispatched to the LSU and all of its operands are available, the **dstx** is queued in a vector-touch queue (VTQ) in the next cycle. There are four data stream engines within the VTQ—data stream 0 uses engine VT0 within the VTQ, data stream 1 uses VT1, and so forth.

The operation of a VT data stream engine does not consume any dispatch or completion resources. A VT is an asynchronous line-fetch or line-touch engine that can prefetch data in units of 32-byte cache blocks by inserting touch requests into the normal load/store pipeline.

After the **dstx** is queued in the VTQ, the VTQ begins to unroll the stream into 32-byte line touches. As early as the third cycle after the LSU sends its request to the VTQ, the VTQ could make its first line-fetch touch request to the data cache.

Note that a data stream engine bases its accesses on effective addresses. This means that each line fetch within a stream accesses the data MMU simultaneously with the L1 data cache and performs a normal translation. There are no arbitrary address boundaries that affect the progress of a given stream.

In addition, if a VTQ line touch accesses a page whose translation does not reside in the data MMU, a table search operation is performed to load that PTE into the data TLB. The TLB is non-blocking during a VTQ-initiated table search operation, meaning that normal loads and stores can hit in the TLB (and in the data cache) during the table search. For details on a table search operation, see [Section 5.5.2.1, “Conditions for a Page Table Search Operation.”](#)

7.1.2.3.1 Stream Engine Tags

The opcodes for the **dstx** instructions are shown in [Table 7-4](#).

Table 7-4. Opcodes for dstx Instructions

Name	0	5	7	8	9	10	11	15	16	20	21	30	31
dst	0111_11	0	00	STRM	A	B	01_0101_0110	0					
dstst	0111_11	0	00	STRM	A	B	01_0111_0110	0					
dststt	0111_11	1	00	STRM	A	B	010_111_0110	0					
dstt	0111_11	1	00	STRM	A	B	01_0101_0110	0					

The STRM field in the **dstx** instruction designates which of the four data stream engines (VT0, VT1, VT2, or VT3) is used by a given instruction, as described in [Table 7-5](#).

Table 7-5. DST[STRM] Description

Value of STRM Field in dstx Instruction	Data Stream Engines (VTs)
00	VT0
01	VT1
10	VT2
11	VT3

Bits 7 and 8 of the **dstx** opcode are reserved. If bit 7 is set, it is ignored. If bit 8 is set, the VTQ does not queue up the stream and that **dstx** instruction is ignored.

7.1.2.3.2 Speculative Execution and Pipeline Stalls for Data Stream Instructions

Like a load miss instruction or a **dcbt/dcbtst** instruction, a **dstx** instruction is executed speculatively. If the target of a particular **dstx** line fetch is mapped with $G = 1$ (guarded), any reload for that line fetch is under the same constraints as a guarded load. If any of the four data stream engines encounter a TLB miss, all four pause until the **dstx** access that caused the TLB miss is retired from the completion queue or is the oldest instruction in the queue. The **dstx** then initiates a table search operation and completes its current cache access.

If a **dstx** instruction to a given data stream is dispatched and the VTQ is processing a previous **dstx** to the same data stream, the second **dst** to that tag supersedes the first one, but only after the second **dstx** becomes non-branch-speculative; it can still be speculative with respect to interrupts. If a third **dstx** is ready for dispatch while the second is waiting for branch speculation to resolve, instruction dispatch stalls.

7.1.2.3.3 Static/Transient Data Stream Touch Instructions

The AltiVec ISA defines two of the **dstx** instructions as static (**dst** and **dstst**) and two as transient (**dstt** and **dststt**). Static data is likely to have a reasonable degree of locality and is referenced several times or over a reasonably long period of time. Transient data is assumed to have poor locality and is likely to be referenced only a few times over a short period of time.

The e600 core supports both static and transient memory-access behavior. The **lvxl** and **stvxl** instructions are interpreted as transient data accesses.

7.1.2.3.4 Relationship with the **sync/tblsync** Instructions

If a **sync** instruction is executed while a **dstx** is in progress, the following happens for each of the four VTs:

- Any cache line fetch in progress continues until that single cache line refill has completed.
- The VTQ pauses and does not continue to its next line-fetch location.
- When all other necessary conditions are met in the machine, the **sync** instruction is completed.
- The **dstx** resumes with cache accesses/reloads to the next line-fetch location.

The effect of the **sync** is a short pause in **dstx** operation. Code sequences that are truly intended to quiet the machine, like those used to enter reduced-power states, must use **dss/dssall** followed by a **sync** instruction to kill outstanding transactions initiated by **dstx** instructions. Refer to [Section 7.1.2.3.8, “Differences Between **dst/dstt** and **dstst/dststt** Instructions,”](#) for more details on the **dstx** and **dss/dssall** instructions.

Note that a **tlbsync** instruction affects the VTQ identically to a **sync** instruction with the additional effect that an outstanding VTQ-initiated table search operation is canceled when a **tlbsync** is dispatched to the LSU.

7.1.2.3.5 Data Stream Termination

If one of the conditions in [Table 7-6](#) is determined to be true when a given line fetch of a **dstx** stream is translated, the entire **dstx** stream is terminated. Note that this can occur in the middle of many line fetches for a **dstx** stream.

If the condition involves address translation and the **dstx** stream specifies an access that would cross into another page, the processor does not attempt to continue the **dstx** stream at those new pages if it had an opportunity to fully translate the access.

Table 7-6. The **dstx Stream Termination Conditions**

Conditions
Successfully reached end of stream
The dstx stream is still speculative with respect to program flow, and the control unit issues a cancel due to a mispredicted branch or interrupt.
Another dstx instruction to this stream tag is executed, and this new dstx is non-speculative with respect to branch prediction.
A dss instruction to this stream tag is completed.
Current line fetch caused a table search operation that did not find a matching entry in the page table.

Table 7-6. The `dstx` Stream Termination Conditions (continued)

Conditions
Current line fetch is translated as cache-inhibited.
Current line fetch is translated as write-through and the stream is a touch-for-store.
Current line fetch is translated to direct-store space ($SR[T] = 1$).
Current line fetch is to a protected page.
L1 data cache is locked or disabled.
The processor has encountered a condition that causes a machine check interrupt.

Note that asserting \overline{sreset} does not terminate a `dstx` stream.

7.1.2.3.6 Line Fetch Skipping

When an interrupt occurs, the e600 core terminates any `dstx`-initiated table search operations and pauses the stream engine that initiated the table search. In this situation, the line fetch of the `dst` that caused the table search is effectively dropped and any translation interrupt that would have terminated the stream had the table search operation completed does not occur. Instead, the engine attempts the next line fetch when the stream resumes. This, in effect, causes a skip of one line fetch in the stream engine.

Also note that the execution of a `tlbsync` instruction cancels any `dstx`-initiated table search operations in progress, which can cause a line fetch skip.

7.1.2.3.7 Context Awareness and Stream Pausing

Stream accesses can take place only when data translation is enabled ($MSR[DR] = 1$), and when the processor is in the same privilege state as it was when the `dstx` instruction was executed.

If the privilege level setting changes or if data translation is disabled, the stream engine suspends generation of new accesses. Any outstanding transactions initiated before the pause (like cache refills and MPX bus activity) finish normally. The stream engine resumes when translation is again enabled and the privilege level again matches the level in place when the `dstx` instruction for that stream was executed.

7.1.2.3.8 Differences Between `dst/dstt` and `dstst/dststt` Instructions

The only difference between touch-for-load (`dst/dstt`) and touch-for-store (`dstst/dststt`) streams is that touch-for-load streams are subdivided into line fetches that are treated identically to individual `dcbt` fetches, while touch-for-store streams are subdivided into line fetches that are treated identically to individual `dcbst` fetches.

Note that if a touch-for-store stream instruction is mapped to a write-through page, that stream is terminated. The use of the touch-for-store streams is not recommended when store-miss merging is enabled, which is the default case. See [Section 3.4.4.4, “Data Cache Block Store \(dcbst\),”](#) for further details on store-miss merging.

Although the e600 core implements touch-for-store stream instructions, use of these instructions is not recommended because it can degrade performance.

7.1.2.4 dss and dssall Instructions

The Data Stream Stop instruction (**dss**) is never executed speculatively. Instead, **dss** instructions flow into a four-entry **dss** queue (DSSQ) in which one entry is dedicated to each possible tag. If another **dss** is dispatched with a tag that matches a non-completed but valid DSSQ entry, that new **dss** remains in a hold queue and waits for the previous **dss** in the DSSQ to be completed.

If a subsequent **dstx** is queued in the VTQ, it cancels an older **dss** entry in the DSSQ (for the same tag). When a given DSSQ entry completes, the valid bit for the VTQ entry corresponding to that tag is immediately cleared.

If a **dssall** instruction is executed, the DSSQ queues all four queue entries in order to terminate all four VT streams when the **dssall** instruction is the oldest. The **dssall** opcode differs from **dss** in that bit 6 (the A field) is set and bits 7–10 are ignored.

Note that line fetches in progress for a given **dstx** stream are not canceled by the **dss** instruction. Only subsequent line fetches are prevented. To ensure that all line fetches from a **dstx** are completed, a **sync** instruction must be issued after the **dss** instruction.

7.1.2.5 Java Mode, NaNs, Denormalized Numbers, and Zeros

This section describes the e600 core floating-point behavior for various special-case data types. The descriptions cover both Java and non-Java modes (see [Section 7.1.1.4, “Vector Status and Control Register \(VSCR\),”](#) for setting Java/non-Java mode), including the following:

- Denormalization for all instructions
- NaNs, denormalized numbers, and zeros for compare, min, and max e600 core operations
- Zero and NaN data for round-to-float integral operations

Note the following:

- The e600 core defaults to Java mode.
- The e600 core handles NaNs the same way regardless of Java or non-Java mode.
- The e600 core handles most denormalized numbers in Java mode by taking a trap to interrupt 0x01600 (AltiVec assist interrupt) but, for some instructions the e600 core can produce the exact result without trapping.

[Table 7-7](#) describes denormalization instructions.

Table 7-7. Denormalization for AltiVec Instructions

Instruction	Input Denormalization Detected		Output Denormalization Detected	
	Java	Non-Java	Java	Non-Java
vaddfp, vsubfp, vmaddfp, vnmsubfp	Trap (unless result is a NaN) ¹	Input treated as correctly signed zero	Trap	Result squashed to correctly signed zero
vrefp	Trap	Denormalized number squashed to zero, returning +/-∞	Trap	Result squashed to zero
vrsqrtefp	Trap	Denormalized number squashed to zero, returning +/-∞	Never produces a denormalized number	Never produces a denormalized number

Table 7-7. Denormalization for AltiVec Instructions (continued)

Instruction	Input Denormalization Detected		Output Denormalization Detected	
	Java	Non-Java	Java	Non-Java
vlogefp	Trap	Denormalized number squashed to zero, returning $-\infty$	Never produces a denormalized number	Never produces a denormalized number
vexptefp	Result is +1.0	Input squashed to zero, output result is +1.0	Trap	Result squashed to zero
vcfux, vcfxsx	Never detects denormalized numbers			
vctxsx, vctuxs	Trap ¹	Output result is 0x0	Never produces a denormalized number	Never produces a denormalized number

¹ May change in the future to produce an IEEE Std. 754 default result in hardware instead of trapping. If the instruction has a denorm operand that would produce a NaN result, the e600 core returns the NaN result and does not cause an AltiVec assist interrupt.

Table 7-8 describes the behavior of the vector floating-point compare, min, and max instructions in non-Java mode.

Table 7-8. Vector Floating-Point Compare, Min, and Max in Non-Java Mode

vA	vB	vminfp	vmaxfp	vcmpgftfp	vcmpgefp	vcmpqfp	vcmpbfp	
							LE	GE
NaN_A	—	QNaN_A	QNaN_A	False	False	False	0	0
—	NaN_B	QNaN_B	QNaN_B	False	False	False	0	0
+Den_A	−B	−B	+Zero	True	True	False	0	0
−Den_A	−B	−B	−Zero	True	True	False	0	0
+Den_A	+B	+Zero	+B	False	False	False	1	1
−Den_A	+B	−Zero	+B	False	False	False	1	1
−A	+Den_B	−A	+Zero	False	False	False	1	0
−A	−Den_B	−A	−Zero	False	False	False	1	0
+A	+Den_B	+Zero	+A	True	True	False	0	1
+A	−Den_B	−Zero	+A	True	True	False	0	1
+Den_A/+Zero	+Den_B/+Zero	+Zero	+Zero	False	True	True	1	1
+Den_A/+Zero	−Den_B/−Zero	−Zero	+Zero	False	True	True	1	1
−Den_A/−Zero	+Den_B/+Zero	−Zero	+Zero	False	True	True	1	1
−Den_A/−Zero	−Den_B/−Zero	−Zero	−Zero	False	True	True	1	1

Table 7-9 describes the behavior of the same instructions in Java mode.

Table 7-9. Vector Floating-Point Compare, Min, and Max in Java Mode

vA	vB	vminfp	vmaxfp	vcmpgtfp	vcmpgefp	vcmppeqfp	vcmpbfp	
							LE	GE
NaN_A	—	QNaN_A	QNaN_A	False	False	False	0	0
—	NaN_B	QNaN_B	QNaN_B	False	False	False	0	0
+Den_A	−B	−B	+Den_A	True	True	False	0	0
−Den_A	−B	−B	−Den_A	True	True	False	0	0
+Den_A	+B	+Den_A	+B	False	False	False	1	1
−Den_A	+B	−Den_A	+B	False	False	False	1	1
−A	+Den_B	−A	+Den_B	False	False	False	1	0
−A	−Den_B	−A	−Den_B	False	False	False	1	0
+A	+Den_B	+Den_B	+A	True	True	False	0	1
+A	−Den_B	−Den_B	+A	True	True	False	0	1
+Den_A	±Zero	±Zero	+Den_A	True	True	False	0	1
−Den_A	±Zero	−Den_A	±Zero	False	False	False	1	0
±Zero	+Den_B	±Zero	+Den_B	False	False	False	1	1
±Zero	−Den_B	−Den_B	±Zero	True	True	False	0	0
−Den_A	+Den_B	−Den_A	+Den_B	False	False	False	1	Result depends on input operands
+Den_A	−Den_B	−Den_B	+Den_A	True	True	False	0	
−Den_A	−Den_B	Result depends on input operands					0	
+Den_A	+Den_B	Result depends on input operands					1	

Table 7-10 describes the behavior of round-to-integer instructions in non-Java mode.

Table 7-10. Round-to-Integer Instructions in Non-Java Mode

vB Sign	vB exponent	Instruction			
		vrfin	vrfiz	vrfip	vrfim
neg	127 > exp > 24	vB	vB	vB	vB
	23 > exp > 0	Round towards nearest	Truncate fraction	Round towards +∞	Round towards −∞
	Exp = −1	Round to nearest	−Zero	−Zero	−1.0
	−2 > exp > −126	−Zero	−Zero	−Zero	−1.0
	Input is denormalized	−Zero	−Zero	−Zero	−Zero
	Input is zero	−Zero	−Zero	−Zero	−Zero

Table 7-10. Round-to-Integer Instructions in Non-Java Mode (continued)

vB Sign	vB exponent	Instruction			
		vrfin	vrfiz	vrfip	vrfim
pos	input is zero	+Zero	+Zero	+Zero	+Zero
	Input is denormalized	+Zero	+Zero	+Zero	+Zero
	$-126 < \text{exp} < -2$	+Zero	+Zero	+1.0	+Zero
	$\text{exp} = -1$	Round towards nearest	+Zero	+1.0	+Zero
	$0 < \text{exp} < 23$	Round towards nearest	Truncate fraction	Round towards $+\infty$	Round towards $-\infty$
	$24 < \text{exp} < 126$	vB	vB	vB	vB

Table 7-11 describes round-to-integer instructions in Java mode. Note that round-to-integer instructions never produce denormalized numbers.

Table 7-11. Round-to-Integer Instructions in Java Mode

vB Sign	vB Exponent	Instruction			
		vrfin	vrfiz	vrfip	vrfim
neg	$127 > \text{exp} > 24$	vB	vB	vB	vB
	$23 > \text{exp} > 0$	Round towards nearest	Truncate fraction	Round towards $+\infty$	Round towards $-\infty$
	$\text{Exp} = -1$	Round to nearest	-Zero	-Zero	-1.0
	$-2 > \text{exp} > -126$	-Zero	-Zero	-Zero	-1.0
	Input is denormalized	Trap	Trap	Trap	Trap
	Input is zero	-Zero	-Zero	-Zero	-Zero
pos	Input is zero	+Zero	+Zero	+Zero	+Zero
	Input is denormalized	Trap	Trap	Trap	Trap
	$-126 < \text{exp} < -2$	+Zero	+Zero	+1.0	+Zero
	$\text{Exp} = -1$	Round towards nearest	+Zero	+1.0	+Zero
	$0 < \text{exp} < 23$	Round to nearest	Truncate fraction	Round To $+\infty$	Round To $-\infty$
	$24 < \text{exp} < 126$	vB	vB	vB	vB

The e600 core detects underflows and production of denormalized numbers on vector float results before rounding, not after. Future versions of the *AltiVec Technology Programming Environments Manual* may reflect this ordering.

7.1.2.6 AltiVec Instructions with Unique Behaviors

The e600 reciprocal estimate for powers of 2 using the **vrefp** instruction is exact. For example, in the e600 core $vrefp(+2.0) = +0.50$.

For the $1/\sqrt{x}$ estimate instruction **vsqrtefp**, the e600 core does not round the least-significant bit of the mantissa.

Additionally, the **vsr** and **vsl** instructions are executed by the vector permute unit on the e600 core.

7.1.2.7 AltiVec Instruction Sequencing

The e600 core implements the AltiVec execution unit as four subunits: simple (VIU1), complex (VIU2), float (VFPU), and permute (VPU). The four-entry AltiVec issue queue (VIQ) can issue up to two instructions to two of the four AltiVec subunits (VIU1, VIU2, VFPU, and VPU) per cycle. For example, the e600 core can issue both a vector simple and a vector complex instruction simultaneously. [Table 7-12](#) lists some high-level microarchitectural features of the AltiVec implementation on the e600 core.

Table 7-12. e600 Core AltiVec Features

Microarchitectural Feature	e600 Core
Available vector execution units	2-issue to any 2 vector units (VIU1, VPU, VIU2, VFPU)
VIU1 Execution Unit Timings (Latency-Throughput)	1-1
VIU2 Execution Unit Timings (Latency-Throughput)	4-1
VFPU Execution Unit Timings (Latency-Throughput)	4-1
VPU Execution Unit Timings (Latency-Throughput)	2-1

[Table 7-13](#) shows where e600 core AltiVec instructions are executed.

Table 7-13. Locations of e600 Core Instruction Execution

Mnemonic	Execution Unit
mfvscr	VFPU
mtvscr	VFPU
vcmpbfp[.]	VFPU
vcmpeqfp[.]	VFPU
vcmpgefp[.]	VFPU
vcmpgtfp[.]	VFPU
vmaxfp	VFPU
vminfp	VFPU
vsl	VPU
vsr	VPU

7.2 AltiVec Technology and the Cache Model

The e600 core uses a unified LSU to load and store operands into the GPRs, FPRs, and VRs. The core's high-bandwidth memory subsystem supports anticipated AltiVec workloads.

The core memory subsystem features summarized in the following sections combine to provide high bandwidth.

The following list summarizes features of the e600 cache implementation that affect the AltiVec implementation:

- The 32-Kbyte, 8-way set associative L1 data cache is fully non-blocking.
 - The 128-bit interface is designed to support AltiVec load/store operations.
 - It supports both MRU (most recently used) and LRU (least recently used) vector loads.
 - New castout and modified bits support **lvx/stvx** LRU operations
 - The allowance of different cache states between the L1 and L2 caches eliminates the need to allocate or update the state in the L2 when a transient (**dstst** or **stvxl**) store occurs to a block that is marked as shared in the L2. In this case, the LLQ treats the L2 block as invalid for stores.
- Pseudo LRU (PLRU) replacement algorithm for L1 cache
- Random replacement for L2 cache
- Support for AltiVec LRU instructions. LRU instructions are described in [Section 7.1.2.1, “LRU Instructions.”](#)
- Support for AltiVec transient instructions. Transient instructions are described in [Section 7.1.2.2, “Transient Instructions and Caches.”](#)

7.3 AltiVec and the Interrupt Model

Only the four following interrupts can result from execution of an AltiVec instruction:

- An AltiVec unavailable interrupt occurs when executing any non-stream AltiVec instruction with $MSR[VEC] = 0$. After this interrupt occurs, execution resumes at offset 0x00F20 from the base physical address indicated by $MSR[IP]$. This interrupt does not occur for data streaming instructions (**dst(t)**, **dstst(t)**, and **dss**). Also note that **VRSAVE** is not protected by this interrupt which is consistent with the *AltiVec Programming Environments Manual*. Thus, any access to the **VRSAVE** register does not cause an interrupt when $MSR[VEC] = 0$.
- A DSI interrupt occurs only if an AltiVec load or store operation encounters a protection violation or a page fault (does not find a valid PTE during a table search operation).
- An AltiVec assist interrupt may occur if an AltiVec floating-point instruction detects denormalized data as an input or output in Java mode.
- AltiVec loads and stores—Note that AltiVec loads and stores that are not quad word-aligned take an alignment interrupt. If either of these conditions is encountered, a re-write of the alignment interrupt routines in software is required.

7.4 AltiVec and the Memory Management Model

The AltiVec functionality in the e600 core affects the MMU model in the following ways:

- A data stream instruction (**dst(t)** or **dstst(t)**) can cause table search operations to occur after the instruction is retired.
- MMU interrupt conditions can cause a data stream operation to abort.
- Aborted VTQ-initiated table search operations can cause a line fetch skip.
- Execution of a **tlbsync** instruction can cancel an outstanding table search operation for a VTQ.

Data stream touch instructions may use either of the two translation mechanisms as specified by the PowerPC architecture—segment/page or BAT. For more information, see [Chapter 5, “Memory Management.”](#)

7.5 AltiVec Technology and Instruction Timing

AltiVec computational instructions are executed in the four independent pipelined AltiVec execution units. The VPU has a two-stage pipeline, the VIU1 has a one-stage pipeline, and the VIU2 and VFPU have four-stage pipelines. As many as 10 AltiVec instructions can be executing concurrently.

The AltiVec technology defines additional data streaming instructions to help improve throughput. Those instructions are described in [Section 7.1.2.3, “Data Stream Touch Instructions.”](#) A complete description of the AltiVec instruction timing is provided in [Chapter 6, “Instruction Timing.”](#)

Chapter 8

Core Interface

This chapter provides a very general description of the e600 core interface, which is the interface between the e600 core and the integrating device. Because most of the behavior of the core interface is not directly programmable, or even visible, to the user, this chapter does not attempt to describe all aspects of the core interface or even the most important core interface signals.

Instead it describes only those aspects of the core interface that are configurable or that provide status information through the programming interface. It provides a table of those signals that are referenced in other chapters to offer a clearer understanding of how the core is integrated as part of a larger device.

8.1 Signal Overview

A subset of the selected internal MPX bus core signals is grouped as follows:

- **Interrupts/Resets**—These signals include the external interrupt signal, checkstop signals, and both soft reset and hard reset signals. They are used to interrupt and, under various conditions, to reset the core.
- **Core status and control**—These signals indicate the state of the core. They include the time-base enable, machine quiesce control, and power management signals.
- **Clock control**—These signals determine the system clock frequency and provide a flexible clocking scheme that allows the processor to operate at an integer multiple of the system clock frequency. They are also used to synchronize multiprocessor systems.
- **Test interface**—The JTAG (IEEE Std. 1149.1a-1993) interface and the common on-chip processor (COP) unit provide a serial interface to the core for performing board-level boundary-scan interconnect tests. The test data input (*tdi*) and test data output (*tdo*) scan ports are used to scan instructions as well as data into the various scan registers for JTAG operations. The scan operation is controlled by the test access port (TAP) controller which in turn is controlled by the test mode select (*tms*) input sequence. The scan data is latched in at the rising edge of test clock (*tck*).
- **Master address bus**—These signals provide information about the type of transfer, such as whether the transaction is bursted, global, write-through, or cache-inhibited. A signal is also provided that indicates if a transfer error occurred.

Note that this chapter does not attempt to characterize MPX bus behavior. [Figure 8-1](#) shows the subset of core interface signals that are described in this chapter.

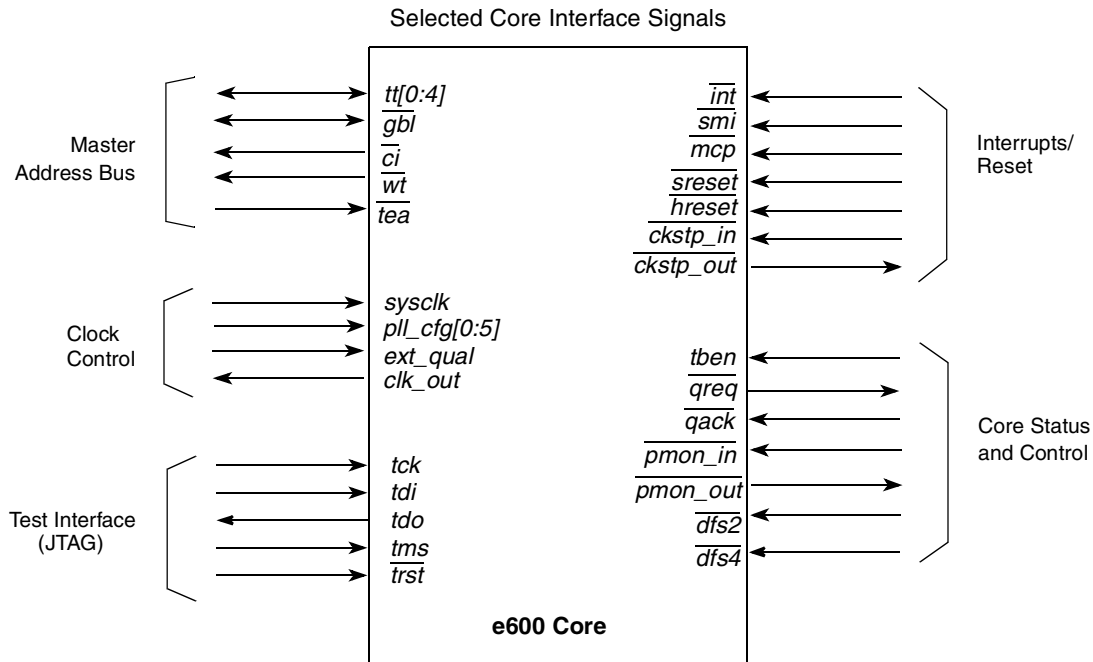


Figure 8-1. Core Interface Signals

8.1.1 Signal Descriptions

Table 8-1 lists selected internal signals of the e600 core interface in alphabetical order and describes the meaning of each signal when it is detected as asserted.

NOTE

A bar over a signal name indicates that the signal is active low—for example, \overline{int} (interrupt) and \overline{sreset} (soft reset). Active-low signals are referred to as asserted (active) when they are low and negated when they are high. Signals that are not active low, such as tdo (JTAG test data output) are referred to as asserted when they are high and negated when they are low.

Table 8-1. Summary of Selected Internal Signals

Signal	Signal Name and Meaning when Asserted	No. of Signals	I/O
\overline{ci}	Cache-inhibited. Indicates that a single-beat transfer is not cached, reflecting the setting of the I bit for the block or page that contains the address of the current transaction (except during certain data cache, memory synchronization, TLB management, and external control operations). \overline{ci} is also asserted for reads and writes if the L1 cache is disabled.	1	O
$\overline{ckstp_in}$	Checkstop in. Indicates that the e600 core must terminate operation by internally gating off all clocks and releasing all outputs (except $\overline{ckstp_out}$) to the high-impedance state. Once $\overline{ckstp_in}$ has been asserted, it must remain asserted until the system has been reset. $\overline{ckstp_in}$ is not maskable.	1	I
$\overline{ckstp_out}$	Checkstop out. Indicates that the e600 core has detected a checkstop condition and has ceased operation.	1	O

Table 8-1. Summary of Selected Internal Signals (continued)

Signal	Signal Name and Meaning when Asserted	No. of Signals	I/O
<i>clk_out</i>	Clock out. Provides a PLL clock output for PLL testing and monitoring. The <i>clk_out</i> signal is provided for testing only.	1	O
$\overline{dfs2}$	Dynamic frequency switching divide-by-two. Divides the core-to-MPX bus clock ratio by two during normal functional operation.	1	I
$\overline{dfs4}$	Dynamic frequency switching divide-by-four. Divides the core-to-MPX bus clock ratio by four during normal functional operation.	1	I
<i>ext_qual</i>	PLL bypass clock. Provides a PLL bypass mode for the system clock.	1	I
\overline{gbl}	Global. As an output, indicates that a transaction is global, reflecting the setting of the M bit for the block or page that contains the address of the current transaction (except during certain data cache, memory synchronization, TLB management, and external control operations as described in Table 3-27). Thus, this transaction must be snooped. As an input, indicates that a transaction must be snooped by the e600 core.	1	I/O
\overline{hreset}	Hard reset. The hard reset (\overline{hreset}) signal must be used at power-on to properly reset the processor. Initiates a complete hard reset operation upon assertion. The hard reset sequence includes the hardware initialization of the e600 core's circuitry. Causes a reset interrupt. Output drivers are released to high impedance and the e600 core is held in an initialized state.	1	I
\overline{int}	Interrupt request. Indicates that the e600 core should take an external interrupt if enabled in the MSR.	1	I
\overline{mcp}	Machine check. Indicates that the e600 core should initiate a machine check interrupt or enter the checkstop state as directed by the MSR.	1	I
<i>pll_cfg[0:5]</i>	PLL configuration. Configure the operation of the PLL and the internal processor clock frequency.	6	I
$\overline{pmon_in}$	Performance monitor in. Indicates that the performance monitor will log a performance monitor pin event, as described in PMC1, bit 7 in Table 10-9 (an event will only be logged if this event is enabled in the performance monitor control registers).	1	I
$\overline{pmon_out}$	Performance monitor out. Indicates that the performance monitor threshold or negative counter condition has been reached.	1	O
\overline{qack}	Quiescent acknowledge. Indicates that all MPX bus activity has terminated or paused and that the e600 core may enter nap or sleep mode.	1	I
\overline{qreq}	Quiescent request. Indicates that the e600 core is requesting all MPX bus activity to terminate or pause so that it may enter a quiescent (low-power nap or sleep) state. Once in this state, the e600 core stops snooping further MPX bus activity.	1	O
\overline{sreset}	Soft reset. Provides a "warm" reset capability, functioning as a non-maskable interrupt. Initiates processing for a reset interrupt, as described in Section 4.6.1, "System Reset Interrupt (0x00100)."	1	I
\overline{smi}	System management interrupt. Indicates that the e600 core should take a system management interrupt if enabled in the MSR.	1	I
<i>sysclk</i>	System clock. The <i>sysclk</i> input is the primary clock input for the e600 core and represents the bus clock frequency for e600 core MPX bus operation. Internally, the e600 core may be operating at an integer or half-integer multiple of the bus clock frequency.	1	I

Table 8-1. Summary of Selected Internal Signals (continued)

Signal	Signal Name and Meaning when Asserted	No. of Signals	I/O
<i>tben</i>	Time base enable. Indicates that the timebase and decremter should continue clocking. This signal functions as a count enable control for the timebase and decremter counter. Note that in addition to the assertion of the <i>tben</i> signal, <i>HID0[TBEN]</i> must also be set in order for the time base and decremter to operate.	1	I
<i>tck</i>	Scan clock. This input should be driven by a free-running clock signal. Input signals to the test access port are clocked-in on the rising edge of <i>tck</i> . Changes to the test access port output signals occur on the falling edge of <i>tck</i> .	1	I
<i>tdi</i>	Serial scan input. The value presented on this signal on the rising edge of <i>tck</i> is clocked into the selected JTAG test instruction or data register.	1	I
<i>tdo</i>	Serial scan output. The contents of the selected internal instruction or data register are shifted out onto this signal on the falling edge of <i>tck</i> .	1	O
\overline{tea}	Transfer error acknowledge. Indicates that a data bus error occurred on the core interface. Causes a machine check interrupt (and possibly causes the processor to enter checkstop state if machine check enable bit is cleared (<i>MSR[ME] = 0</i>)). For more information, see Section 4.6.2.2, "Checkstop State (<i>MSR[ME] = 0</i>)" . For reads, assertion of \overline{tea} does not invalidate data entering the GPRs or the caches.	1	I
<i>tms</i>	Test mode select. Decoded by the internal JTAG TAP controller to distinguish the primary operation of the test support circuitry.	1	I
\overline{trst}	Test reset. Causes asynchronous initialization of the internal JTAG test access port controller. Assures that the JTAG logic does not interfere with the normal operation of the device.	1	I
<i>tt[0:4]</i>	Transfer type. Indicates the type of transfer in progress.	5	I/O
\overline{wt}	Write-through. Indicates that a single-beat write transaction is write-through, reflecting the value of the W bit for the block or page that contains the address of the current transaction (except during certain data cache, memory synchronization, TLB management, and external control operations).	1	O

8.1.2 Configuration Signals Sampled at Reset

[Table 8-2](#) contains a description of the signals sampled for configuration at the negation of \overline{hreset} . Note that throughout this manual, the reset configuration signals are described as being sampled at the negation of *reset*. However, the reset configuration signals are actually sampled 3 clock cycles before the negation of \overline{hreset} .

The values on these signals during reset are interpreted to be logic one or zero, regardless of whether the signal name is defined as active-low. The *pll_cfg[0:5]* signal must be driven high or low during the reset period.

Table 8-2. e600 Reset Configuration Signals

Signal Name(s)	Default	State Meaning
<i>pll_cfg[0:5]</i>	Must be driven	These six signals select the clock frequency ratios used by the PLL of the e600 core.

8.1.3 Reset, Interrupt, Checkstop, and Power Management Signal Interactions

This section describes the hard and soft reset input signals, external interrupts, checkstop operations, and power management signal interactions. See [Chapter 4, “Interrupts,”](#) for more information on the interrupts caused by these signals.

8.1.3.1 Reset Inputs

The e600 core has two reset inputs, described as follows:

- \overline{hreset} (hard reset)—The \overline{hreset} signal is used for power-on reset sequences, or for situations in which the e600 core must go through the entire cold start sequence of internal hardware initialization.
- \overline{sreset} (soft reset)—The soft reset input provides warm reset capability. This input can be used to avoid forcing the e600 core to complete the cold start sequence.

When either reset input negates, the processor attempts to fetch code from the system reset interrupt vector. The vector is located at offset 0x0100 from the interrupt prefix (MSR[IP]). The MSR[IP] bit is set when \overline{hreset} negates.

8.1.3.2 External Interrupts

The external interrupt input signals (\overline{int} , \overline{smi} and \overline{mcp}) of the e600 core force the core to take the external interrupt vector or the system management interrupt vector if the MSR[EE] is set, or the machine check interrupt if the MSR[ME] and the HID1[EMCP] bits are set.

8.1.3.3 Checkstops

The e600 core has two checkstop input signals— $\overline{ckstp_in}$ (nonmaskable) and \overline{mcp} (enabled when MSR[ME] is cleared and HID1[EMCP] is set)—and a checkstop output ($\overline{ckstp_out}$) signal. If $\overline{ckstp_in}$ or \overline{mcp} is asserted, the e600 core halts operations by gating off all internal clocks. The e600 core asserts $\overline{ckstp_out}$ if $\overline{ckstp_in}$ is asserted.

If $\overline{ckstp_out}$ is asserted by the e600 core, it has entered the checkstop state, and processing has halted internally. The $\overline{ckstp_out}$ signal can be asserted for various reasons including receiving a *tea* (transfer error acknowledge) signal and detection of parity errors external to the core. For more information about the checkstop state, see [Section 4.6.2.2, “Checkstop State \(MSR\[ME\] = 0\).”](#)

All non-test output signals are disabled during a checkstop.

8.1.3.4 Power Management Signals

This section describes the e600 core's support for power management. The system quiescence control signals (\overline{qreq} and \overline{qack}) allow the processor to enter the nap or sleep low-power states and bring MPX bus activity to a quiescent state in an orderly fashion.

Prior to entering the nap or sleep power state, the e600 core asserts the \overline{qreq} signal. This signal allows the system to terminate or pause any MPX bus activities that are normally snooped. When the system is ready

to enter the system quiesce state, it asserts the \overline{qack} signal. At this time the e600 core may enter the nap or sleep power-saving state. When the e600 core is in the quiescent state, it stops snooping MPX bus activity.

While the e600 core is in the nap state, the system power controller can enable snooping by the e600 core by negating the $qack$ signal for at least eight bus clock cycles, after which the e600 core is capable of snooping MPX bus transactions. The reassertion of \overline{qack} following the snoop transactions causes the e600 core to reenter the nap power state. See [Chapter 9, “Power and Thermal Management,”](#) for more information on the power-saving modes of the e600 core.

Once the e600 core has made a request to enter the nap power-saving state, the \overline{qreq} signal may be negated on any clock cycle to service an internal interrupt (such as a decremter or time base interrupt).

8.1.4 IEEE Std. 1149.1a-1993 Interface

The e600 boundary-scan interface is a fully-compliant implementation of the IEEE 1149.1a-1993 standard. This section briefly describes the e600 IEEE Std. 1149.1a-1993 (JTAG) interface.

8.1.4.1 JTAG/COP Interface

The e600 core has extensive test capability including the following:

- Debug control/observation (COP)
- Boundary scan (interface designed to comply with IEEE Std. 1149.1a-1993 (JTAG))
- Support for manufacturing test
- Support for the following standard JTAG instructions:
 - BYPASS
 - EXTEST
 - SAMPLE/PRELOAD
 - CLAMP
 - HIGHZ

The COP and boundary-scan logic are not used under typical operating conditions. Detailed discussion of the e600 test functions is beyond the scope of this document.

The JTAG/COP interface is shown in [Figure 8-2](#). For more information, refer to *IEEE Standard Test Access Port and Boundary Scan Architecture IEEE Std. 1149-1a-1993*.

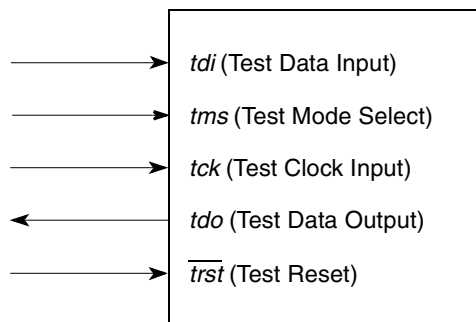


Figure 8-2. IEEE Std. 1149.1a-1993 Boundary-Scan Interface

8.2 e600 Core Interface Summary

8.2.1 MPX Bus Features

The MPX bus has the following features:

- Extended 36-bit address bus plus 5 bits of odd parity (41 bits total)
- 64-bit data bus plus 8 bits of odd parity (72 bits total); a 32-bit data bus mode is not supported
- Support for a four-state (MESI) cache coherence protocol
- Snooping within the core to maintain L1 data cache, and L2 cache coherency for multiprocessing applications and DMA environments
- Support for address-only transfers (useful for a variety of broadcast operations in multiprocessor applications)
- Address pipelining
- Support for up to 16 out-of-order transactions
- Full data streaming
- Support for data intervention in multiprocessor systems

8.2.2 Overview of Core Interface Accesses

The core interface includes address register queues, prioritization logic, and a bus control unit. The core interface latches snoop addresses for snooping in the L1 data cache and the L2 cache, the memory hierarchy address register queues, and the reservation controlled by the Load Word and Reserve Indexed (**lwarx**) and Store Word Conditional Indexed (**stwcx**) instructions. Accesses are prioritized with load operations preceding store operations.

Instructions are automatically fetched from the memory system into the instruction unit where they are dispatched to the execution units at a peak rate of three instructions per clock. Conversely, load and store instructions explicitly specify the movement of operands to and from the integer, floating-point, and AltiVec register files and the memory system.

When the e600 core encounters an instruction or data access, it calculates the effective address and uses the lower-order address bits to check for a hit in the 32-Kbyte L1 instruction and data caches within the core. During L1 cache lookup, the instruction and data memory management units (MMUs) use the higher-order address bits to calculate the virtual address, from which they calculate the physical address (real address). The physical address bits are then compared with the corresponding cache tag bits to determine if a cache hit occurred in the L1 instruction or data cache. If the access misses in the corresponding cache, the transaction is sent to the L1 load miss queue or the L1 store miss queue. L1 load miss queue transactions are sent to the internal 1-Mbyte L2 cache. Store miss queue transactions are queued up in the L2 cache controller. If no match is found in the L2 cache tags, the physical address is used to access system memory.

In addition to the loads, stores, and instruction fetches, the e600 core performs hardware table search operations following TLB misses, L1 and L2 cache castout operations, and cache-line snoop push operations when a modified cache line detects a snoop hit from another bus master.

[Figure 8-3](#) shows a block diagram of the e600 core, including the address path from the execution units and instruction fetcher through the translation logic to the caches and core interface logic. [Figure 8-3](#) shows the organization of the e600 core execution units.

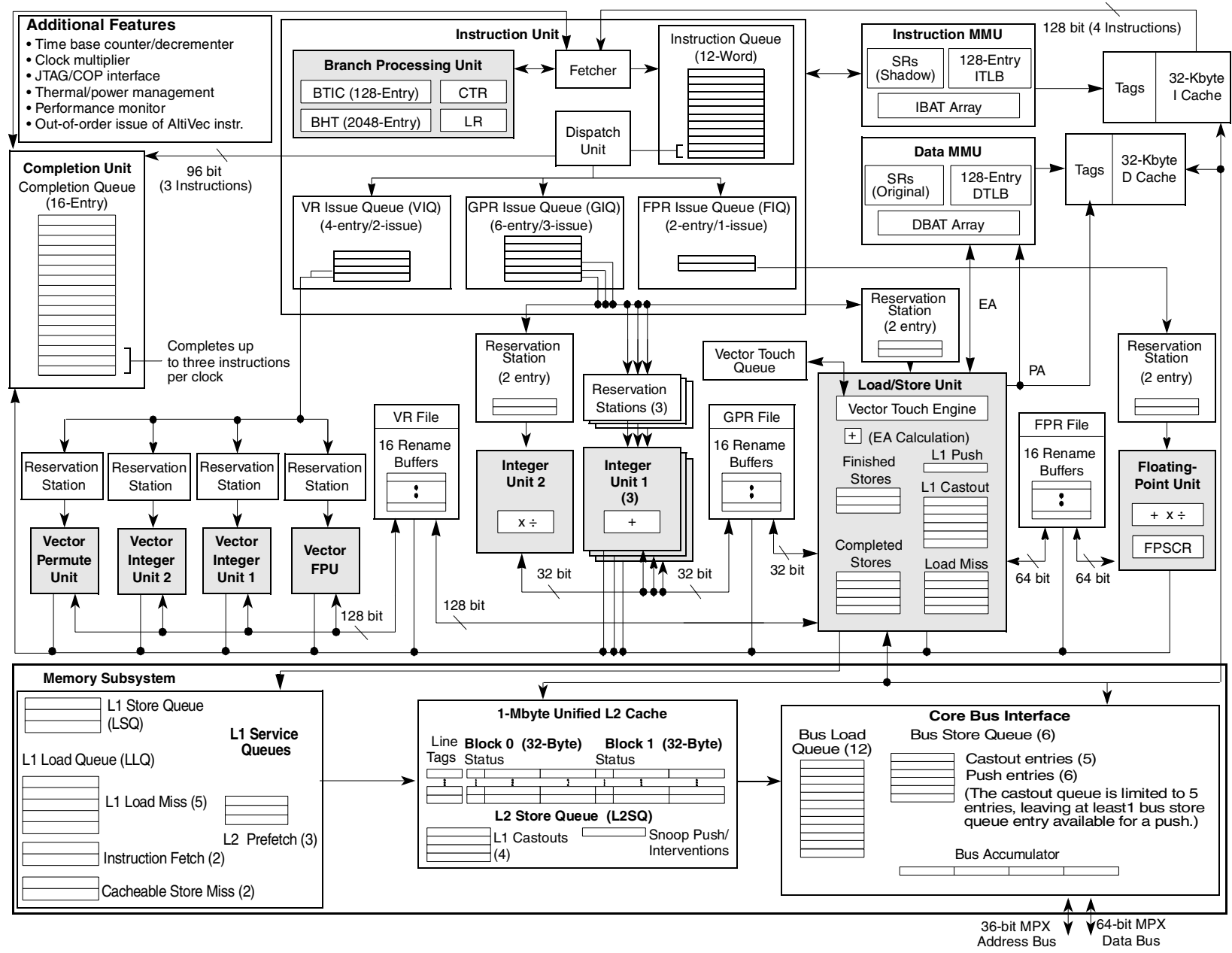


Figure 8-3. e600 Core Block Diagram

The e600 core uses separate address and data buses and a variety of control and status signals, only a subset of which is described in this chapter, for performing reads and writes external to the core. The address bus is 36 bits wide and the data bus is 64 bits wide. The interface is synchronous—all e600 inputs are sampled at and all outputs are driven from the rising edge of the bus clock. The processor runs at a multiple of the bus clock speed.

8.2.3 Summary of L1 Instruction and Data Cache Operation

The e600 core provides independent L1 instruction and data caches. Each cache is a physically-addressed, 32-Kbyte cache with 8-way set associativity. Both caches consist of 128 sets of 8 cache lines, with 32 consecutive bytes in each cache line. The e600 data cache tags are dual-ported and non-blocking, allowing efficient load/store and snoop operations. The e600 core supports a four-state cache coherency protocol that includes Modified (M), Exclusive (E), Shared (S), and Invalid (I) data cache states.

The cache control instructions, **dcbt**, **dcbst**, **dcbz**, **dcbst**, **dcbf**, **dcba**, **dcbi**, and **icbi**, are intended for the management of the local L1 and L2 caches. The e600 core interprets the cache control instructions as if they pertain only to its own caches. These instructions are not intended for managing other caches in the system (except to the extent necessary to maintain coherency). The e600 core snoops all global (\overline{gbl} asserted) cache control instruction broadcasts. Execution of the **dcbz** and **dcba** instructions cause a broadcast on the MPX bus (when $M = 1$) unless the address cache block is found in the exclusive state in the L1 or L2 cache. The **dcbst**, **dcbf**, **dcbi**, and **icbi** instructions cause a broadcast on the MPX bus (when $M = 1$) only if $HID1[ABE]$ is set.

Because the data cache on the e600 core is a write-back primary cache, the predominant type of transaction for most applications is a burst-read memory operation, followed by burst-write memory operations and single-beat (caching-inhibited or write-through) memory read and write operations. Additionally, there can be address-only operations, variants of the burst and single-beat operations (for example, global memory operations that are snooped and atomic memory operations), and address retry activity (for example, when a snooped read access hits a modified line in the cache).

On a cache miss, cache blocks are filled in one 32-byte burst. The burst fill is performed as a critical-double-word-first operation. For the instruction cache, the critical double word is forwarded to the instruction queue as soon as it is available, thus minimizing stalls due to cache fill latency. The instruction cache is not blocked to internal accesses while a load completes, providing for hits under misses. For the data cache, an entire cache block is collected in an accumulator latch before being loaded into the cache. The critical double word is forwarded to the execution units as soon as it is available.

Cache lines are selected for replacement based on a pseudo least-recently-used (PLRU) algorithm. Each time a cache block is accessed, it is tagged as the most recently used way of the set (unless accessed by the AltiVec LRU instructions, see [Section 7.1.2.1, “LRU Instructions”](#)). For every hit in the cache or when a new block is reloaded, the PLRU bits for the set are updated. Data cache replacement selection is performed at reload time, not when a miss occurs. However, instruction cache replacement selection occurs when an instruction cache miss is first recognized—that is, the instruction cache replacement target is selected upon a miss and not at reload.

A data cache block fill is caused by a load miss or write-back store miss in the cache. The cache block that corresponds to the missed address is updated by a burst transfer of the data from the L2 cache or system memory after any necessary coherency actions have completed.

For more information about the interactions of the instruction and data caches and the core interface, see [Section 3.7, “Core Interface.”](#)

8.2.4 L2 Cache Overview

The e600 core features an integrated L2 cache that is a unified (containing instruction and data) 1-Mbyte cache. It is 8-way set associative and organized with 32-byte blocks and two blocks per line. Thus each block in a line shares the same tag, but the valid, modified, and shared bits are independently maintained for each block.

8.2.5 Operation of the Core Interface

Memory accesses can occur in single-beat (1, 2, 3, 4, and 8 bytes), double-beat (16 bytes), and four-beat (32 bytes) burst data transfers. For memory accesses, the address and data buses are independent to support pipelining and split transactions. The core interface can pipeline as many as 16 transactions and supports full out-of-order split-bus transactions. The e600 core bursts out of reset, fetching eight instructions on the MPX bus at a time.

Typically, memory accesses are weakly ordered to maximize the efficiency of the MPX bus without sacrificing coherency of the data. The e600 core allows load operations to bypass store operations (except when a dependency exists). Because the processor can dynamically optimize run-time ordering of load/store traffic, overall performance is improved.

Note that the synchronize (**sync**) and enforce in-order execution of I/O (**eiio**) instructions can be used to enforce strong ordering.

8.2.6 Memory Subsystem Control Register (MSSCR0)

The MSSCR0 control register is used to configure many aspects of the memory subsystem for the e600 core. At power-on reset, functions are set to a default; thus the MSSCR0 should be changed if non-default functionality is required. It is a supervisor-only read/write, implementation-specific register accessed as SPR 1014.

MSSCR0 includes parameters that set the maximum number of transactions that an e600 core can carry in its data transaction queue, alter how the core responds to snoop requests, enable or disable data intervention, and indicate when the core is operating in the address bus drive mode. Other parameters in MSSCR0 control L2 cache prefetching. See [Section 2.1.6.3, “Memory Subsystem Control Register \(MSSCR0\),”](#) for more detailed information about the bits of MSSCR0.

8.2.7 Memory Subsystem Status Register (MSSSR0)

The memory subsystem status register (MSSSR0) acknowledges bus transfer errors and indicates when parity errors are detected in the L2 cache and the data or address buses of the core interface. See [Section 2.1.6.4, “Memory Subsystem Status Register \(MSSSR0\),”](#) for more detailed information about the bits of MSSSR0.

8.2.8 Direct-Store Accesses Not Supported

The e600 core does not support the extended transfer protocol for accesses to the direct-store storage space. The transfer protocol used for any given access is selected by the T bit in the MMU segment registers; if the T bit is set, the memory access is a direct-store access. Any attempt to access instructions or data in a direct-store segment causes the e600 core to take an ISI or DSI interrupt.

8.2.8.1 Address Transfer Attributes

The transfer attributes include the transfer type ($tt[0:4]$), write-through (\overline{wt}), cache inhibit (\overline{ci}), and global (\overline{gbl}) signals. The read claim (RCLAIM) transfer type exists for accesses generated by touch-for-store instructions.

8.2.8.1.1 Transfer Type ($tt[0:4]$) Signals

The transfer type signals ($tt[0:4]$) indicate the type of transaction to be performed. Snooping logic on the integrated device should fully decode the transfer type signals if the \overline{gbl} signal is asserted. Slave devices can sometimes use the individual transfer type signals without fully decoding the group. [Table 8-3](#) describes the MPX bus specification transfer encodings and the behavior of the e600 core as a bus master. RCLAIM is used to identify touch-for-store instructions on the MPX bus. The effect of the RCLAIM transaction is to establish exclusive ownership of a cache line without marking that cache line as modified in the requesting processor's cache.

Table 8-3. Transfer Type Encodings

Generated by e600 Core as Bus Master		$tt0$	$tt1$	$tt2$	$tt3$	$tt4$	MPX Bus Specification	
Type	Source						Command	Type
Address only	dcbst	0	0	0	0	0	Clean block	Address only
Address only	dcbf	0	0	1	0	0	Flush block	Address only
Address only	sync	0	1	0	0	0	sync	Address only
Address only	dcba, dcbz ¹	0	1	1	0	0	Kill block	Address only
Address only	eieio	1	0	0	0	0	eieio	Address only
Address only	tlbie	1	1	0	0	0	TLB invalidate	Address only
N/A	N/A	0	0	0	0	1	lwarx reservation set	Address only
N/A	N/A	0	0	1	0	1	Reserved	—
Address only	tlbsync	0	1	0	0	1	tlbsync	Address only
Address only	icbi	0	1	1	0	1	icbi	Address only
N/A	N/A	1	X	X	0	1	Reserved	—
Single-beat write or burst	Caching-inhibited or write-through store	0	0	0	1	0	Write-with-flush	Single-beat write or burst

Table 8-3. Transfer Type Encodings (continued)

Generated by e600 Core as Bus Master		<i>tt0</i>	<i>tt1</i>	<i>tt2</i>	<i>tt3</i>	<i>tt4</i>	MPX Bus Specification	
Type	Source						Command	Type
Burst (non- \overline{gbl})	Cast-out, dcbf , dcbi , dcbst push, or snoop copyback	0	0	1	1	0	Write-with-kill	Burst
Single-beat read or burst	Data load or instruction fetch	0	1	0	1	0	Read	Single-beat read or burst
N/A	N/A	1	0	0	1	0	Write-with-flush-atomic ^{2, 3}	Single-beat write
N/A	N/A	1	0	1	1	0	Reserved	N/A
Burst	lwarx ³	1	1	0	1	0	Read-atomic	Single-beat read or burst
Burst	stwcx. ³	1	1	1	1	0	Read-with-intent-to-modify-atomic	Burst
Burst	Store miss	0	1	1	1	0	Read-with-intent-to-modify	Burst
N/A	N/A	0	0	0	1	1	Reserved	—
N/A	N/A	0	0	1	1	1	Reserved	—
N/A	N/A	0	1	0	1	1	Read-with-no-intent-to-cache	Single-beat read or burst
Burst	dstst , dststt , or dcbtst	0	1	1	1	1	Read claim (RCLAIM)	Burst
N/A	N/A	1	X	X	1	1	Reserved	—

¹ The core can also issue a kill block if a series of stores results in an entire cache line being modified; see [Section 3.1.2.3, “Store Gathering/Merging,”](#) for details.

² Not generated by the e600 core, but snooped.

³ Caching-inhibited **lwarx** and caching-inhibited or write-through **stwcx.** cause DSI interrupts on the e600 core.

8.2.9 Write-Through (\overline{wt}), Cache Inhibit (\overline{ci}), and Global (\overline{gbl}) Signals

In general, the e600 core provides the \overline{wt} , \overline{ci} , and \overline{gbl} signals to indicate the status of a transaction target as determined by the WIMG bit settings during address translation by the MMU. There are interrupts, as described in [Section 3.7.2, “Transfer Attributes.”](#)

During write operations, the \overline{wt} signal reflects the write-through status for the transaction as determined by the MMU address translation. It is also asserted for burst writes due to **dcbf** (flush) and **dcbst** (clean) instructions and for snoop pushes.

The e600 core also provides the \overline{ci} transfer attribute. This signal reflects the caching-inhibited status of the transaction as determined by the MMU address translation (WIMG bits). Note that if the L1 data cache is disabled, all data accesses are treated as caching-inhibited and \overline{ci} is asserted regardless of the WIMG bit settings.

The e600 core provides the additional transfer attribute \overline{gbl} . This signal indicates the coherency requirements (global or non-global) for the transaction as determined by the MMU address translation. As an output, it indicates that a transaction is global. As an input, it indicates that a transaction must be snooped by the e600 core.



Chapter 9

Power and Thermal Management

This chapter describes the power management facilities as they are implemented in devices that contain the e600 core. The scope of this chapter is limited to the features of the core only. Additional power management capabilities associated with a device that integrates this core (referenced as the integrated device throughout the chapter) are documented separately in the device reference manual.

9.1 Overview

A complete power management scheme for a system using the e600 core requires the support of logic in the integrated device. The core provides software with a way to signal a need for power management to the integrated device. It also provides a signal interface that the integrated device can use to transition the core into its different power management states.

9.2 Power Management Signals

Table 9-1 summarizes the power management signals of the e600 core.

Table 9-1. Power Management Signals of the e600 Core

Signal	I/O	Description
<i>nap</i>	O	Reflect the state of the corresponding HID0[NAP, SLEEP] bits (if MSR[POW] = 1); both must be set for the respective output (<i>nap</i> or <i>sleep</i>) to assert. These signals do not affect the core's power-down state, but indicate to the integrated device that software has made power management requests. Integrated device logic may use these signals to affect device-level power state, which in turn may affect the core power state (signaled through <i>halt</i> , and <i>stop</i>).
<i>sleep</i>	O	
<i>halt</i>	I	Asserted by integrated device logic after detecting either <i>nap</i> or <i>sleep</i> asserted. The following actions are initiated as a result and cause the core to enter core-halted state: <ul style="list-style-type: none"> • Suspend instruction fetching. • Complete all previously fetched instructions. • When the instruction pipeline is empty, the core asserts the <i>halted</i> output. The core clock continues running. Negating <i>halt</i> returns the core to full-power state. If it is negated before the core has entered core-halted state, the negation may not be recognized.
<i>halted</i>	O	Asserted by the core when it reaches core-halted state. Indicates to the integrated device logic that it is safe to power-down; that is, no data is lost on transition to the core-stopped state.

Table 9-1. Power Management Signals of the e600 Core (continued)

Signal	I/O	Description
<i>stop</i>	I	Asserted by integrated device logic to initiate the required actions that cause the core to go from core-halted to core-stopped state (as described in Table 9-2). Negating <i>stop</i> returns the core to core-halted state. Once asserted, <i>stop</i> must not be negated until after the core has entered the core-stopped state; otherwise the negation may not be recognized. For power management purposes, <i>stop</i> must be asserted only while the core is in the core-halted state.
<i>stopped</i>	O	Asserted by the core anytime the internal functional clocks of the core are stopped (for example after integrated device logic asserts <i>stop</i>).

9.3 Core Power Management States

The notion of nap, sleep, and deep sleep modes (or states) pertains to the integrated device as a whole. As shown in Figure 9-1, an integrated device may interpret the assertion of *nap* and *sleep* to trigger actions that affect the device-level power state, which in turn may use the *halt* and *stop* inputs to determine how the core transitions between the core-specific power states: full on, core halted, and core stopped.

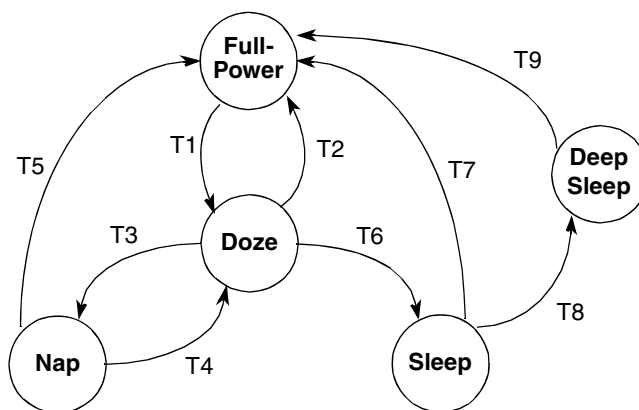


Figure 9-1. e600 Core Power Management State Diagram

The transitions are described in Table 9-2.

Table 9-2. Power Management State Transitions

Transition	Cause
T1	HID0[NAP] or HID0[SLEEP] and MSR[POW] set in the e600 core; integrated device logic asserts <i>halt</i> ; the e600 becomes halted and asserts <i>halted</i>
T2	External interrupt, \overline{smi} , \overline{sreset} , \overline{hreset} , \overline{mcp} , decremter interrupt
T3	HID0[NAP] and MSR[POW] are set, the integrated device logic asserts <i>stop</i> , the bus and SRAM are idle.
T4	The integrated device logic negates <i>stop</i> , signaling a pending snoop operation
T5	External interrupt, \overline{smi} , \overline{sreset} , \overline{hreset} , \overline{mcp} , decremter interrupt
T6	HID0[SLEEP] and MSR[POW] are set, the integrated device logic asserts <i>stop</i> , the bus and SRAM are idle.

Table 9-2. Power Management State Transitions (continued)

Transition	Cause
T7	External interrupt, \overline{smi} , \overline{sreset} , \overline{hreset} , or \overline{mcp}
T8	Disable the PLL.
T9	Resume SYSCLK if disabled. Assert \overline{hreset}

To wake from a power-saving mode using an external interrupt or decremter interrupt, the user should make sure that MSR[EE] is set. Waking the e600 from deep sleep can be achieved only by asserting hard reset.

A hardware/software combination may be used to enable power management states through external asynchronous interrupts. Such a hardware interrupt transfers the program flow to the system management interrupt (SMI) handler, which then sets the appropriate power-saving mode. A decremter interrupt can be used to return to full power from nap states after a predetermined duration.

9.3.1 Full-Power State

Full-power state is the default power state after power-up and \overline{hreset} . All functional units are operating and all clocks are enabled. If dynamic power management mode (DPM) is enabled (HID0[DPM] = 1), some idle functional units automatically enter a lower-power state without affecting performance, software execution, or external hardware. DPM functions independently of the power management states and need not be enabled for any of doze, nap, sleep, or deep sleep to be utilized.

9.3.2 Doze State

All functional units including instruction dispatch are disabled in doze state. Each functional unit continues to receive clocks and therefore consume power. The bus interface will recognize and process snoop transactions on the external bus. The interrupt interface will recognize the assertion of external interrupts. The decremter unit will continue to function if enabled, and decremter exceptions will be recognized.

The power savings due to disabling instruction dispatch is minimal. The doze state is mainly used as a temporary state while transitioning the processor:

1. To the nap or sleep states from full-power state.
2. Out of nap state to perform snooping.

9.3.2.1 Entering Doze State

In full-power state, software can set HID0[NAP] or HID0[SLEEP] and MSR[POW]. After the setting is selected, instruction dispatch is automatically halted only after the integrated device logic asserts *halt*. All outstanding instructions are completed. Completion includes arbitrating for the external bus. All outstanding bus transactions finish all outstanding address and data tenures. Once the core is quiesced, it will assert the *halted* signal, indicating that it is in doze state (core halted state).

9.3.2.2 Exiting Doze State

Any \overline{int} , \overline{smi} , \overline{sreset} , or \overline{mcp} assertion as well as a decremter exception wakes the e600 from doze state and puts it into full-power state within a few core clock cycles. Assertion of \overline{hreset} will cause the processor to exit doze state but will require the necessary number of clock cycles to initialize the processor. See the hardware specifications for details.

9.3.3 Nap State

All functional units including instruction dispatch are disabled in nap state. Clocks are disabled to each functional unit. The bus interface will not recognize and process snoop transactions on the external bus. If snooping is required, the core must transition to doze state. The interrupt interface will recognize the assertion of external interrupts. The decremter unit will continue to function if enabled, and decremter exceptions will be recognized. The PLL is operational.

9.3.3.1 Entering Nap State

If the processor is in doze state, and $HID0[NAP]$ is set, the integrated device logic can assert *stop* to the processor. Once *stop* is asserted to the processor, all functional unit clocks are disabled within four integrated host system clocks. *halted* remains asserted by the processor. *stop* must remain asserted by the integrated host system to keep the processor in nap state.

9.3.3.2 Exiting Nap State

Any \overline{int} , \overline{smi} , \overline{sreset} , or \overline{mcp} assertion, or decremter exception, wakes the e600 from nap state and puts it into full-power state within a few core clock cycles. The e600 negates *halted* and *stopped* when transitioning to full-power state. Assertion of \overline{hreset} will cause the e600 to exit nap state but will require the necessary number of clock cycles to initialize. See the hardware specifications for details.

9.3.3.3 Snooping in Nap State

Snooping is disabled in nap state, however, by negating *stop* to the e600 the system can transition to doze state. The e600 requires four bus cycles to transition from nap to doze state before the system may drive a snoop transaction on the bus. *stop* should remain negated while snooping the processor. If the e600 asserts *artry* as a result of a snoop, the system must wait for a clean response before reasserting *stop*. The e600 may also assert \overline{hit} in response to the snoop. It is up to the system to wait for the clean response and/or handle any data tenure resulting from the snoop. Multiple snoops may be attempted while *stop* is negated.

9.3.4 Sleep State

All functional units including instruction dispatch are disabled in sleep state. Clocks are disabled to each functional unit. The bus interface will not recognize and process snoop transactions on the external bus. No transition to the doze state exists; therefore snooping is not possible. Caches must be flushed to guarantee cache coherency if other system components access memory while the e600 core is sleeping. The interrupt interface will recognize the assertion of external interrupts. The decremter unit clocks are disabled. Because the time base and decremter are disabled while the e600 is in sleep state, the time base contents must be updated from an external timebase after exiting sleep state if accurate time-of-day is

required. The PLL is operational: it remains locked and running but no clocks propagate to functional units.

Sleep state has limited power savings improvements compared to nap state, and lacks snooping and decremter functionality. Sleep state is beneficial if the goal is to transition to deep sleep state.

9.3.4.1 Entering Sleep State

If the processor is in doze state, and `HID0[SLEEP]` is set, the integrated device logic can assert *stop* to the processor. Once *stop* is asserted to the processor, all functional unit clocks are disabled within four integrated host system clocks. *halted* remains asserted by the processor. *stop* must remain asserted by the integrated host system to keep the processor in sleep state.

9.3.4.2 Exiting Sleep State

Any \overline{int} , \overline{smi} , \overline{sreset} , or \overline{mcp} assertion wakes the e600 core from sleep state and puts it into full-power state within a few processor cycles. The decremter is disabled and therefore is not available to wake the processor from sleep state. The e600 negates *halted* and *stopped* when transitioning to full-power state. Assertion of \overline{hreset} will cause the processor to exit sleep state but will require the necessary number of clock cycles to initialize the processor. See the hardware specifications to determine the required number of clock cycles.

9.3.4.3 Snooping in Sleep State

Snooping is disabled in sleep state. Negating *stop* has no effect. Caches should be flushed of any modified data before entering sleep state.

9.3.5 Deep Sleep State

From sleep state, the e600 can achieve further power savings by disabling the PLL or the SYSCLK source.

9.3.5.1 Entering Deep Sleep State

From sleep state, the system can configure the PLL to the 'PLL off state' (`pll_cfg[0:5] = 0x3C`).

9.3.5.2 Exiting Deep Sleep State

Waking the e600 from deep sleep can be achieved only by asserting \overline{hreset} . First, *sysclk* should be resumed if it was disabled. Second, `pll_cfg` should be reprogrammed to the appropriate value. Third, the hard reset power up sequence should be followed for restarting and relocking the PLL. Refer to power-on reset instructions for the time required for the PLL to achieve lock.

9.3.6 Power Management Software Considerations

Because the e600 core is a three-issue core with out-of-order execution capability, care must be taken in how the power management state is entered. Furthermore, all outstanding MPX bus operations must be completed before entering nap and sleep states. Normally, during system configuration time, one of these power management states is selected by setting the appropriate `HID0` bit. Later on, the state is invoked by

setting MSR[POW]. To ensure a clean transition into and out of a power-management state, the **mtmsr** that sets the POW bit must be preceded by the proper cache flushes, instruction cache disable followed by **isync**, TLB invalidates, and then **dssall** and **sync** instructions. The sleep mode entry sequence is as follows:

```

mtHID0 (NAP | SLEEP)
...
...
dssall
... cache flushes... (1)
... instruction cache disable ... (2)
isync (3)
... TLB invalidates ... (4)
loop    sync
        mtmsr[POW = 1]
        isync
        b loop
    
```

The **dssall** instruction is needed to kill any outstanding stream touch instructions not killed by a **sync**.

In addition to the power-management states, dynamic power management (enabled by setting HID0[DPM]) automatically stops clocking individual internal functional units whenever they are idle. The integrated logic may similarly stop clocking to idle device-level blocks.

9.4 Power Management Control Bits

Although the core can signal power management requests through the bits shown in [Table 9-3](#), core power management is controlled by the integrated device, which may provide additional ways to put the core into a power-saving state. Interlocks between the core and the integrated device prevent data loss that could occur if one part of the system powered down before the other had time to prepare.

Table 9-3. Core Power Management Control Bits

Bit	Description
MSR[POW]	Must be set for HID0[NAP, SLEEP] to cause assertion of <i>nap</i> and <i>sleep</i> to system logic.
HID0[NAP]	If MSR[POW] = 1, signals power management logic to initiate device nap mode. The core enters core-stopped state (with its time base enabled) after integrated device logic asserts <i>stop</i> .
HID0[SLEEP]	If MSR[POW] = 1, signals power management logic to initiate device sleep mode. The core remains in core-stopped state and stops its time base after integrated device logic negates <i>tben</i> .

NOTE

The e600 does not implement its own doze, nap, and sleep modes. The core-halted and core-stopped states may correlate to the integrated device’s doze and nap, sleep, and deep sleep modes, but the e600 cannot be placed into core-halted or core-stopped state without interaction with system integration logic.

9.5 Power Management Protocol

The e600 outputs the *nap* and *sleep* signals to the integrated device logic, which controls power states both for the device as a whole and for the core (namely the core-halted and core-stopped states). Figure 9-2 shows how device integration logic would typically respond to *nap* and *sleep* and control the core's power state through the *halt/halted*, and *stop/stopped* signal pairs.

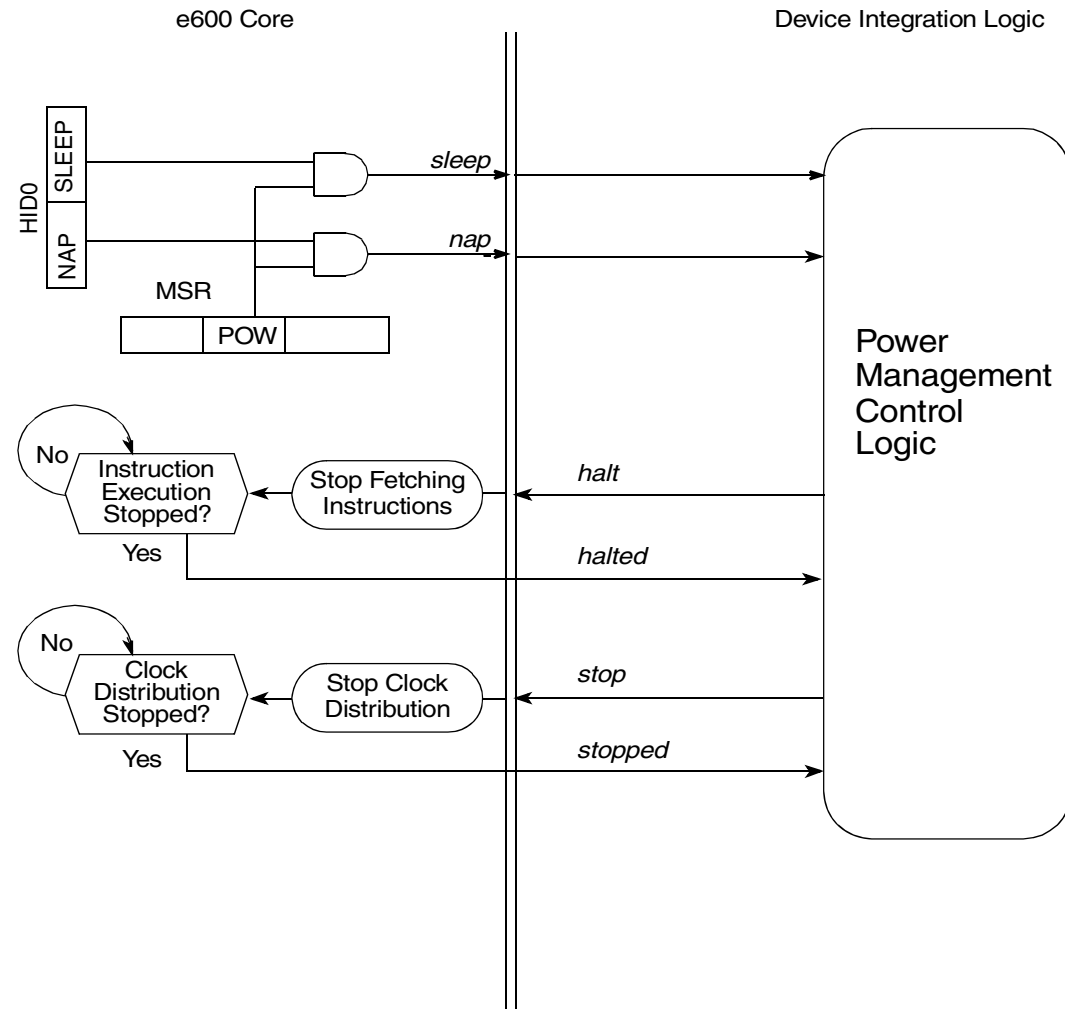


Figure 9-2. Example Core Power Management Handshaking

9.6 Interrupts and Power Management

In *halted* or *stopped* states, the core does not recognize and process interrupts. The power management logic of the integrated device must monitor all external interrupt requests to detect interrupt requests. Upon sensing an interrupt request, the integrated device ordinarily negates *stop* and *halt* to restore the core to full-power state, allowing it to service the interrupt request.

- When the processor exits a low-power state, MSR[POW], which enables power-down requests, is cleared without being automatically restored.
- Assuming that the system entered a low-power state in response to the assertion of *nap* or *sleep*, the integrated device's power management logic must recognize that these outputs remain asserted for some time after the core is restored to full-power state (due to the normal latency of restarting internal clock distribution and initiating the interrupt request), and then negate as the interrupt is serviced.

9.6.1 Dynamic Frequency Switching (DFS)

The dynamic frequency switching (DFS) feature has the ability to divide the core-to-MPX bus ratio by two or four during normal functional operation by setting the HID1[DFS2, DFS4] bits. The frequency change occurs in one clock cycle, and no idle waiting period is required to switch between modes. This feature may or may not be used in combination with voltage derating by a system to reduce the power consumption of the core.

For example, a core operating at 1-GHz internal frequency, configured to an 8:1 core-to-MPX bus ratio at power-on reset, can dynamically change the core-to-MPX bus ratio to 4:1 by setting the DFS2 bit. The core frequency would be reduced to 500 MHz without ever cycling the core through hard reset. The applied MPX bus clock frequency also does not need to change for this power reduction step.

NOTE

DFS is not available for all core-to-MPX bus ratios configurable during hard reset.

The minimum PLL ratio implemented by DFS is 2:1. DFS can be used in conjunction with the Nap and Sleep modes and it is possible to enter and exit these modes while DFS is enabled. In addition, the e600 core allows hardware control of DFS through the $\overline{dfs2}$ and $\overline{dfs4}$ signals, making it possible to exit nap or sleep mode and return to full-power mode with DFS enabled. These signals can be used in conjunction with or in lieu of software control, and can also be used to disable DFS mode completely.

9.6.1.1 Snooping Restrictions

The core requires a minimum of 5 core cycles to process a snoop and generate a response after indicating an address bus transfer has begun. As a result, if the core frequency is less than five times the MPX bus frequency, the system must extend the address tenure of all transactions that are snooped by the core. For core-to-MPX bus frequency ratios of 2:1 and 2.5:1, assertion that the address tenure should be terminated must be delayed a minimum of 2 bus cycles; for core-to-MPX bus frequency ratios of 3:1, 3.5:1, 4:1, and 4.5:1, this address acknowledge be delayed a minimum of 1 bus cycle. [Table 9-4](#) summarizes the required system delay for ratios less than 5:1.

Table 9-4. Required System Address Acknowledge Delay for Ratios < 5:1

Dynamic PLL Ratio	Required Delay
2:1	2 cycles
2.5:1	
3:1	1 cycle
3.5:1	
4:1	
4.5:1	

Since the core requires 5 internal clock cycles to provide the correct response to a snoop on the MPX bus, delaying address acknowledge assures that the correct snoop response will be asserted by the core.

The proper sequencing for changing delay is to:

1. Modify the system to delay address acknowledge.
2. Alter the HID1[DFS2, DFS4] bits to choose the ratio < 5:1.
3. Operate at the lower frequency.
4. Clear the DFS bits, and
5. Modify the system to remove the delay.

9.7 Instruction Cache Throttling

The e600 core provides an instruction cache throttling mechanism to reduce the instruction execution rate. When used with dynamic power management, instruction cache throttling provides the system designer with a flexible way to control device temperature while allowing the processor to continue operating. However, note that performance does degrade with instruction throttling and that entering short bursts of nap or sleep is a superior method for reducing thermal output and power consumption.

The instruction cache throttling mechanism simply reduces the instruction dispatch rate. Normally, as many as three instructions are dispatched in each cycle. For thermal management, the e600 core provides a supervisor-level instruction cache throttling control register (ICTC), shown in [Figure 9-3](#). The instruction dispatch rate is reduced by setting ICTC[E] to enable instruction cache throttling and writing a nonzero value into ICTC[INTERVAL]. The ICTC[INTERVAL] field specifies the minimum number of cycles between instructions being dispatched. When the ICTC[E] bit is clear, the e600 core dispatches instructions normally. The overall junction temperature reduction results from dynamic power management of execution units when the core is idle between instruction dispatches; thus, instruction cache throttling does not provide thermal reduction unless HID0[DPM] = 1. Note that during instruction cache throttling, the PLL configuration remains unchanged.

Reserved

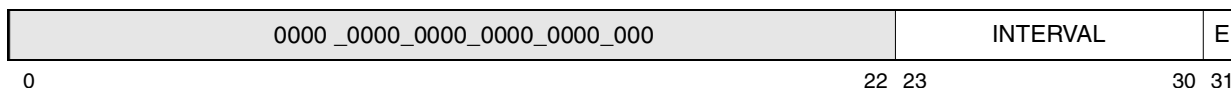


Figure 9-3. Instruction Cache Throttling Control Register (ICTC)

Table 9-5 describes ICTC fields.

Table 9-5. ICTC Field Descriptions

Bits	Name	Description
0–22	—	Reserved, should be cleared.
23–30	INTERVAL	Instruction forwarding interval expressed in core clocks. When throttling is enabled, the interval field specifies the minimum number of cycles between instructions being dispatched. (The e600 core dispatches one instruction every INTERVAL cycle.) The minimum interval for throttling control is 2 cycles. 0x00, 0x01, 0x02 One instruction dispatches every 2 core clocks 0x03 One instruction dispatches every 3 core clocks ... 0xFF One instruction dispatches every 255 core clocks
31	E	Enable instruction throttling. 0 Instructions dispatch normally. 1 Only one instruction dispatches every INTERVAL cycles.

Chapter 10

Performance Monitor

The PowerPC architecture defines an optional performance monitor facility that provides the ability to monitor and count predefined events such as core clocks, misses in the instruction cache, data cache, or L2 cache, types of instructions dispatched, mispredicted branches, and other occurrences. The count of such events (that may be an approximation) can be used to trigger the performance monitor exception. Note that some earlier processors implemented the performance monitor facility before it was defined by the PowerPC architecture.

The performance monitor can be used for the following:

- To increase system performance with efficient software, especially in a multiprocessing system—Memory hierarchy behavior can be monitored and studied in order to develop algorithms that schedule tasks (and perhaps partition them) and that structure and distribute data optimally.
- To characterize processors—Some environments may not be easily characterized by a benchmark or trace.
- To help system developers bring up and debug their systems

AltiVec Technology and the Performance Monitor

The AltiVec technology features do not affect the basic implementation of the performance monitor. However, the performance monitor provides the ability to count the following AltiVec operations:

- Instructions executed by the individual AltiVec execution units
- Completed AltiVec load instructions
- Cycles during which the VIU1, VIU2, VFPU, and VPU had a valid dispatch but invalid operands
- VFPU traps that can be generated in Java mode
- Completed **mtvscr** and **mtvrsave** instructions
- Times that the VSCR saturate bit is set
- Completed of **dss** and **dssall** instructions
- **dstx** instruction event counts—dispatches, misses, refreshes, suspensions, premature cancellations, and resumptions
- Cycle counts when the VALU has a valid **mfvscr** dispatch but cannot execute it because it is not at the bottom of the completion queue (CQ)

10.1 Overview

The performance monitor uses the following resources defined by the PowerPC architecture:

- The performance monitor mark bit in the MSR (MSR[PMM]). This bit identifies programs to be monitored.
- The privilege level bit in the MSR (MSR[PR]). This bit identifies the mode the processor is in (supervisor or user).
- Special-purpose registers (SPRs):
 - The performance monitor counter registers (PMC1–PMC6) are 32-bit counters used to count the times a software-selectable event has occurred. PMC5 and PMC6 are used to count events generated by the core memory subsystem. UPMC1–UPMC6 provide user-level read access to these registers.
 - The monitor mode control registers (MMCR0–MMCR2). Control fields in the MMCR n registers select events to be counted, determine whether performance monitor interrupts are caused by a time-base register transition, maintain counter overflow, and specify the conditions under which counting is enabled. UMMCR0–UMMCR2 provide user-level read access to these registers.
 - The sampled instruction address register (SIAR) contains the effective address of the last instruction that completes before the performance monitor interrupt is generated. USIAR provides user-level read access to the SIAR.
 - Note that an **mtspr** or **mfspr** SDAR or USDAR instruction causes a program interrupt in the e600 core.
- The performance monitor interrupt follows the normal PowerPC interrupt model and has a defined interrupt vector offset (0x00F00). Its priority is below the trace interrupt and above the AltiVec unavailable interrupt.

10.2 Performance Monitor Interrupt

The performance monitor provides the ability to generate a performance monitor interrupt triggered by an enabled condition or event. This interrupt is triggered by an enabled condition or event defined as follows:

- A PMC n register overflow condition occurs:
 - MMCR0[PMC1CE] and PMC1[OV] are both set
 - MMCR0[PMC n CE] and PMC n [OV] are both set ($n > 1$)
- A time-base event—MMCR0[TBEE] = 1 and the TBL bit specified in MMCR0[TBSEL] changes from 0 to 1

MMCR0[PMXE] must be set for any of these conditions to signal a performance monitor interrupt.

Although the performance monitor interrupt may occur with MSR[EE] = 0, the interrupt is not taken until MSR[EE] = 1.

As a result of a performance monitor interrupt being generated, the performance monitor saves in the SIAR the effective address of the last instruction completed before the interrupt is taken. Note that SIAR is not updated if performance monitor counting has been disabled by setting MMCR0[0].

The priority of the performance monitor interrupt is below the trace interrupt and above the AltiVec unavailable interrupt. See [Section 4.2, “e600 Core Interrupt Recognition and Priorities,”](#) for a list of interrupt priorities.

Interrupt handling for the performance monitor interrupt is described in [Section 4.6.13, “Performance Monitor Interrupt \(0x00F00\).”](#)

10.2.1 Performance Monitor Signals

The $\overline{pmon_in}$ signal is used by the performance monitor event $MMCR0[PMC1SEL] = 7$ (0b000_0111) to count the number of times the $\overline{pmon_in}$ signal transitions from negated to asserted. Note that this event is enabled in the performance monitor control registers (MMCR0, or MMCR1, $PMCn$) and must be enabled in order for this event to be monitored.

The $\overline{pmon_out}$ signal is asserted by the performance monitor when a performance monitor threshold or negative counter condition has been reached, whether or not performance monitor interrupts are enabled; that is, the setting of $MMSR0[PMXE]$ does not affect the function of this signal.

10.2.2 Using Timebase Event to Trigger or Freeze a Counter or Generate an Interrupt

The use of the trigger and freeze counter conditions depends on these enabled conditions and events. When $MMCR0[TBEE]$ (timebase enable event) is 1, a timebase transition is generated to the performance monitor if the TBL bit specified in $MMCR0[TBSEL]$ changes from 0 to 1. Timebase transition events can be used to freeze the counters ($MMCR0[FCECE]$), trigger the counters ($MMCR0[TRIGGER]$), or generate an interrupt ($MMCR0[PMXE]$).

Changing the bits specified in $MMCR0[TBSEL]$ while $MMCR0[TBEE]$ is set may cause a false 0 to 1 transition that signals the specified action (freeze, trigger, or interrupt) to occur immediately.

10.3 Performance Monitor Registers

The following sections describe the registers used by the performance monitor. Note that reading and writing to the performance monitor SPRs do not synchronize the processor.

An explicit synchronization instruction, such as **sync**, should be placed before and after an **mf spr** or **mt spr** of one of these registers to guarantee an accurate count.

10.3.1 Performance Monitor Special-Purpose Registers

The performance monitor incorporates the SPRs listed in [Table 10-1](#) and [Table 10-2](#). The supervisor-level registers in [Table 10-1](#) are accessed through the **mt spr** and **mf spr** instructions.

Table 10-1. Performance Monitor SPRs—Supervisor Level

SPR Number	spr[5–9] spr[0–4]	Register Name
944	0b11101 10000	Monitor mode control register 2—MMCR2
945	0b11101 10001	Performance monitor counter register 5—PMC5
946	0b11101 10010	Performance monitor counter register 6—PMC6
951	0b11101 10111	Breakpoint address mask register—BAMR
952	0b11101 11000	Monitor mode control register 0—MMCR0
953	0b11101 11001	Performance monitor counter register 1—PMC1
954	0b11101 11010	Performance monitor counter register 2—PMC2
955	0b11101 11011	Sampled instruction address register—SIAR
956	0b11101 11100	Monitor mode control register 1—MMCR1
957	0b11101 11101	Performance monitor counter register 3—PMC3
958	0b11101 11110	Performance monitor counter register 4—PMC4

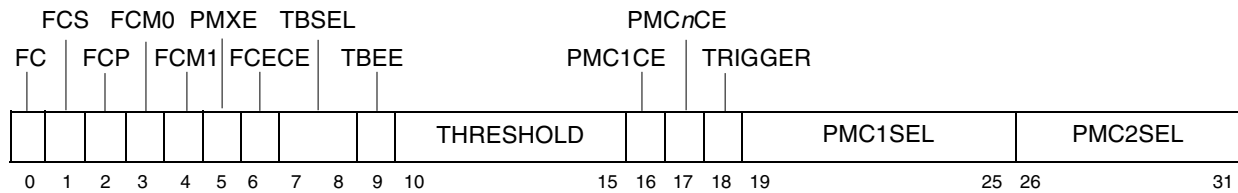
The user-level registers in [Table 10-2](#) are read-only and are accessed with the **mf spr** instruction. Attempting to write to one of these registers in either supervisor or user mode causes a program interrupt.

Table 10-2. Performance Monitor SPRs—User Level (Read-Only)

SPR Number	spr[5–9] spr[0–4]	Register Name
928	0b11101 00000	User monitor mode control register 2—UMMCR2
929	0b11101 00001	User performance monitor counter register 5—UPMC5
930	0b11101 00010	User performance monitor counter register 6—UPMC6
936	0b11101 01000	User monitor mode control register 0—UMMCR0
937	0b11101 01001	User performance monitor counter register 1—UPMC1
938	0b11101 01010	User performance monitor counter register 2—UPMC2
939	0b11101 01011	User sampled instruction address register—USIAR
940	0b11101 01100	User monitor mode control register 1—UMMCR1
941	0b11101 01101	User performance monitor counter register 3—UPMC3
942	0b11101 01110	User performance monitor counter register 4—UPMC4

10.3.2 Monitor Mode Control Register 0 (MMCR0)

The monitor mode control register 0 (MMCR0), shown in [Figure 10-1](#) is a 32-bit SPR provided to specify events to be counted and recorded. If the state of MSR[PR] and MSR[PMM] matches a state specified in MMCR0, then counting is enabled; see [Section 10.4, “Event Counting,”](#) for further details. The MMCR0 can be accessed only in supervisor mode. User-level software can read the contents of MMCR0 by issuing an **mf spr** instruction to UMMCR0, described in [Section 10.3.2.1, “User Monitor Mode Control Register 0 \(UMMCR0\).”](#)


Figure 10-1. Monitor Mode Control Register 0 (MMCR0)

This register is automatically cleared at power-up. Reading this register does not change its contents. [Table 10-3](#) describes the fields of MMCR0.

Table 10-3. MMCR0 Field Descriptions

Bits	Name	Description
0	FC	Freeze counters 0 The PMCs are incremented (if permitted by other MMCR bits). 1 The PMCs are not incremented (performance monitor counting is disabled). The processor automatically sets this bit when an enabled condition or event occurs and MMCR0[FCECE] = 1. Note that SIAR is not updated if performance monitor counting is disabled.
1	FCS	Freeze counters in supervisor mode 0 The PMCs are incremented (if permitted by other MMCR bits). 1 The PMCs are not incremented if MSR[PR] = 0.
2	FCP	Freeze counters in user mode 0 The PMCs are incremented (if permitted by other MMCR bits). 1 The PMCs are not incremented if MSR[PR] = 1.
3	FCM1	Freeze counters while mark = 1 0 The PMCs are incremented (if permitted by other MMCR bits). 1 The PMCs are not incremented if MSR[PMM] = 1.
4	FCM0	Freeze counters while mark = 0 0 The PMCs are incremented (if permitted by other MMCR bits). 1 The PMCs are not incremented if MSR[PMM] = 0.
5	PMXE	Performance monitor interrupt enable 0 Performance monitor interrupts are disabled. 1 Performance monitor interrupts are enabled until a performance monitor interrupt occurs; at that time, MMCR0[PMXE] is automatically cleared. Software can clear PMXE to prevent performance monitor interrupts. Software can also set PMXE and then poll it to determine whether an enabled condition or event occurred.
6	FCECE	Freeze counters on enabled condition or event 0 The PMCs are incremented (if permitted by other MMCR bits). 1 The PMCs are incremented (if permitted by other MMCR bits) until an enabled condition or event occurs when MMCR0[TRIGGER] = 0, at that time MMCR0[FC] is set. If the enabled condition or event occurs when MMCR0[TRIGGER] = 1, FCECE is treated as if it were 0. The use of the trigger and freeze counter conditions depends on the enabled conditions and events described in Section 10.2, "Performance Monitor Interrupt."

Table 10-3. MMCR0 Field Descriptions (continued)

Bits	Name	Description
7–8	TBSEL	<p>Time-base selector. Selects the time-base bit that can cause a time-base transition event (the event occurs when the selected bit changes from 0 to 1).</p> <p>00 TBL[31] 01 TBL[23] 10 TBL[19] 11 TBL[15]</p> <p>Time-base transition events can be used to periodically collect information about processor activity. In multiprocessor systems in which the TB registers are synchronized among processors, time-base transition events can be used to correlate the performance monitor data obtained by the several processors. For this use, software must specify the same TBSEL value for all the processors in the system. Because the time-base frequency is implementation-dependent, software should invoke a system service program to obtain the frequency before choosing a value for TBSEL.</p>
9	TBEE	<p>Time-base event enable</p> <p>0 Time-base transition events are disabled. 1 Time-base transition events are enabled. A time-base transition is generated to the performance monitor if the TB bit specified in MMCR0[TBSEL] changes from 0 to 1. Time-base transition events can be used to freeze the counters (MMCR0[FCECE]), trigger the counters (MMCR0[TRIGGER]), or signal an interrupt (MMCR0[PMXE]).</p> <p>Changing the bits specified in MMCR0[TBSEL] while MMCR0[TBEE] is enabled may cause a false 0 to 1 transition that signals the specified action (freeze, trigger, or interrupt) to occur immediately.</p>
10–15	THRESHOLD	<p>Threshold. Contains a threshold value between 0 to 63. Two types of thresholds can be counted.</p> <ul style="list-style-type: none"> The first type counts any event that lasts longer than the threshold value and uses MMCR2[THRESHMULT] to scale the threshold value by 2 or 32. The second type counts only the events that exceed the threshold value. This type does not use MMCR2[THRESHMULT] to scale the threshold value (MMCR2[THRESHMULT] = 0). <p>By varying the threshold value, software can obtain a profile of the characteristics of the events subject to the threshold. For example, if PMC1 counts cache misses that the duration exceeds the threshold value, software can obtain the distribution of cache miss durations for a given program by monitoring the program repeatedly using a different threshold value each time.</p>
16	PMC1CE	<p>PMC1 condition enable. Controls whether counter negative conditions due to a negative value in PMC1 are enabled.</p> <p>0 Counter negative conditions for PMC1 are disabled. 1 Counter negative conditions for PMC1 are enabled. These events can be used to freeze counters (MMCR0[FCECE]), trigger counters (MMCR0[TRIGGER]), or signal an interrupt (MMCR0[PMXE]).</p>
17	PMC n CE	<p>PMCn condition enable. Controls whether the counter negative conditions due to a negative value in any PMCn (that is, in any PMC except PMC1) are enabled.</p> <p>0 Counter negative conditions for all PMCns are disabled. 1 Counter negative conditions for all PMCns are enabled. These events can be used to freeze counters (MMCR0[FCECE]), trigger counters (MMCR0[TRIGGER]), or signal an interrupt (MMCR0[PMXE]).</p>

Table 10-3. MMCR0 Field Descriptions (continued)

Bits	Name	Description
18	TRIGGER	<p>Trigger</p> <p>0 The PMCs are incremented (if permitted by other MMCR bits).</p> <p>1 PMC1 is incremented (if permitted by other MMCR bits). The PMCns are not incremented until PMC1 is negative or an enabled condition or event occurs, at that time the PMCns resume incrementing (if permitted by other MMCR bits) and MMCR0[TRIGGER] is cleared. The description of FCECE explains the interaction between TRIGGER and FCECE.</p> <p>Uses of TRIGGER include the following:</p> <ul style="list-style-type: none"> Resume counting in the PMCns when PMC1 becomes negative without causing a performance monitor interrupt. Then freeze all PMCs (and optionally cause a performance monitor interrupt) when a PMCn becomes negative. The PMCns then reflect the events that occurred after PMC1 became negative and before PMCn becomes negative. This use requires the following MMCR0 bit settings: <ul style="list-style-type: none"> –TRIGGER = 1 –PMC1CE = 0 –PMCnCE = 1 –TBEE = 0 –FCECE = 1 –PMXE = 1 (if a performance monitor interrupt is desired) Resume counting in the PMCns when PMC1 becomes negative and cause a performance monitor interrupt without freezing any PMCs. The PMCns then reflect the events that occurred between the time PMC1 became negative and the time the interrupt handler reads them. This use requires the following MMCR0 bit settings: <ul style="list-style-type: none"> –TRIGGER = 1 –PMC1CE = 1 –TBEE = 0 –FCECE = 0 –PMXE = 1 <p>The use of the trigger and freeze counter conditions depends on the enabled conditions and events described in Section 10.2, “Performance Monitor Interrupt.”</p>
19–25	PMC1SEL	PMC1 selector. Contains a code (one of at most 128 values) that identifies the event to be counted in PMC1. See Table 10-9 .
26–31	PMC2SEL	PMC2 selector. Contains a code (one of at most 64 values) that identifies the event to be counted in PMC2. See Table 10-10 .

MMCR0 can be accessed with the **mtspr** and **mfspir** instructions using SPR 952.

10.3.2.1 User Monitor Mode Control Register 0 (UMMCR0)

The contents of MMCR0 are reflected to UMMCR0, that can be read by user-level software. UMMCR0 can be accessed with the **mfspir** instructions using SPR 936.

The MMCR2 register is shown in [Figure 10-3](#).

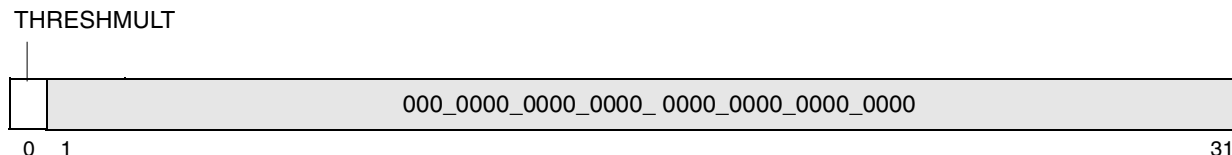


Figure 10-3. Monitor Mode Control Register 2 (MMCR2)

[Table 10-5](#) describes MMCR2 fields.

Table 10-5. MMCR2 Field Descriptions

Bits	Name	Description
0	THRESHMULT	Threshold multiplier. Used to extend the range of the THRESHOLD field, MMCR0[10–15]. 0 Threshold field is multiplied by 2. 1 Threshold field is multiplied by 32.
1–31	—	Reserved

MMCR2 can be accessed with **mtspr** and **mf spr** using SPR 944. User-level software can read the contents of MMCR2 by issuing an **mf spr** instruction to UMMCR2, described in [Section 10.3.4.1, “User Monitor Mode Control Register 2 \(UMMCR2\).”](#)

10.3.4.1 User Monitor Mode Control Register 2 (UMMCR2)

The contents of MMCR2 are reflected to UMMCR2, which can be read by user-level software. UMMCR2 can be accessed with the **mf spr** instructions using SPR 928.

10.3.5 Breakpoint Address Mask Register (BAMR)

The breakpoint address mask register (BAMR), shown in [Figure 10-4](#), is used in conjunction with the events that monitor IABR hits.

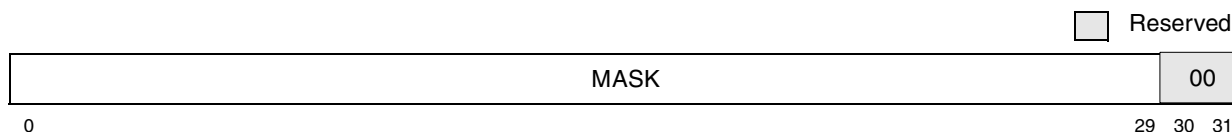


Figure 10-4. Breakpoint Address Mask Register (BAMR)

Table 10-6 describes BAMR fields.

Table 10-6. BAMR Field Descriptions

Bit	Name	Description
0–29	MASK	Used with PMC1 event (PMC1 event 42) that monitor IABR hits The addresses to be compared for an IABR match are affected by the value in BAMR: <ul style="list-style-type: none"> IABR hit (PMC1, event 42) occurs if IABR_CMP (that is, IABR AND BAMR) = instruction_address_compare (that is, EA AND BAMR) IABR_CMP[0–29] = IABR[0–29] AND BAMR[0–29] instruction_addr_cmp[0–29] = instruction_addr[0–29] AND BAMR[0–29] Be aware that breakpoint event 42 of PMC1 can be used to trigger ISI interrupts when the performance monitor detects an enabled overflow. This feature supports debug purposes and occurs only when IABR[30] is set. To avoid taking one of the above interrupts, IABR[30] should be cleared.
30–31	—	Reserved

BAMR can be accessed with **mtspr** and **mfspr** using SPR 951. For synchronization requirements on the BAMR register see [Table 2-37](#).

10.3.6 Performance Monitor Counter Registers (PMC1–PMC6)

PMC1–PMC6, shown in [Figure 10-5](#), are 32-bit counters that can be programmed to generate a performance monitor interrupt when they overflow.

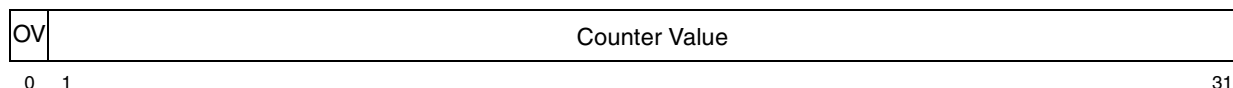


Figure 10-5. Performance Monitor Counter Registers (PMC1–PMC6)

The bits contained in the PMC registers are described in [Table 10-7](#).

Table 10-7. PMC_n Field Descriptions

Bits	Name	Description
0	OV	Overflow. When this bit is set, it indicates that this counter has overflowed and reached its maximum value so that PMC _n [OV] = 1.
1–31	Counter value	Counter value. Indicates the number of occurrences of the specified event.

Counters overflow when the high-order (sign) bit becomes set; that is, they reach the value 2,147,483,648 (0x8000_0000). However, an interrupt is not generated unless both MMCR0[PMXE] and either MMCR0[PMC1CE] or MMCR0[PMCCCE] are also set as appropriate.

Note that the interrupt can be masked by clearing MSR[EE]; the performance monitor condition may occur with MSR[EE] cleared, but the interrupt is not taken until MSR[EE] is set. Setting MMCR0[FCECE] forces counters to stop counting when a counter interrupt or any enabled condition or event occurs. Setting MMCR0[TRIGGER] forces counters PMC_n ($n > 1$), to begin counting when PMC1 goes negative or an enabled condition or event occurs.

Software is expected to use the **mtspr** instruction to explicitly set PMC to non-overflowed values. Setting an overflowed value may cause an erroneous interrupt. For example, if both MMCR0[PMXE] and either MMCR0[PMC1CE] or MMCR0[PMCNCE] are set and the **mtspr** instruction loads an overflow value, an interrupt may be taken without an event-counting having taken place.

The PMC registers can be accessed with the **mtspr** and **mfspir** instructions using the following SPR numbers:

- PMC1 is SPR 953
- PMC2 is SPR 954
- PMC3 is SPR 957
- PMC4 is SPR 958
- PMC5 is SPR 945
- PMC6 is SPR 946

10.3.6.1 User Performance Monitor Counter Registers (UPMC1–UPMC6)

The contents of the PMC1–PMC6 are reflected to UPMC1–UPMC6, which can be read by user-level software. The UPMC registers can be read with the **mfspir** instructions using the following SPR numbers:

- UPMC1 is SPR 937
- UPMC2 is SPR 938
- UPMC3 is SPR 941
- UPMC4 is SPR 942
- UPMC5 is SPR 929
- UPMC6 is SPR 930

10.3.7 Sampled Instruction Address Register (SIAR)

The sampled instruction address register (SIAR) is a supervisor-level register that contains the effective address of the last instruction to complete before the performance monitor interrupt is generated. The SIAR is shown in [Figure 10-6](#).

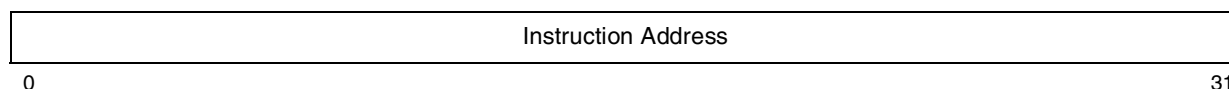


Figure 10-6. Sampled Instruction Address Register (SIAR)

Note that SIAR is not updated:

- if performance monitor counting has been disabled by setting MMCR0[FC] or
- if the performance monitor interrupt has been disabled by clearing MMCR0[PMXE].

SIAR can be accessed with the **mtspr** and **mfspir** instructions using SPR 955.

10.3.7.1 User Sampled Instruction Address Register (USIAR)

The contents of SIAR are reflected to USIAR, which can be read by user-level software. USIAR can be accessed with the **mf spr** instructions using SPR 939.

10.4 Event Counting

Counting can be enabled if conditions in the processor state match a software-specified condition. Because a software task scheduler may switch a processor's execution among multiple processes and because statistics only on a particular process may be of interest, a facility is provided to mark a process. The performance monitor bit, MSR[PMM], is used for this purpose. System software may set this bit when a marked process is running. This enables statistics to be gathered only during the execution of the marked process. The states of MSR[PR] and MSR[PMM] together define a state that the processor (supervisor or user) and the process (marked or unmarked) may be in at any time. If this state matches a state specified in the MMCR (the state in which monitoring is enabled), counting is enabled. [Table 10-8](#) lists the states that can be monitored.

Table 10-8. Monitorable States

MSR[PR]	MSR[PMM]	Process Marked	Process UnMarked	User Mode	Supervisor Mode
0	—	—	—	√	√
1	—	—	—	√	—
—	1	√	—	—	—
—	0	—	√	—	—
0	0	—	√	√	√
0	1	√	—	√	√
1	0	—	√	√	—
1	1	√	—	√	—

In addition, one of two unconditional counting modes may be specified:

- Counting is unconditionally enabled regardless of the states of MSR[PMM] and MSR[PR]. This can be accomplished by clearing MMCR0[0–4].
- Counting is unconditionally disabled regardless of the states of MSR[PMM] and MSR[PR]. This is done by setting MMCR0[0] = 1. Note that SIAR is not updated if MMCR0[0] = 1.

10.5 Event Selection

Event selection is handled through the MMCR_n registers, described in subsequent tables.

- The six event-select fields in MMCR0 and MMCR1 are as follows:
 - MMCR0[PMC1SEL]—PMC1 input selector, 128 events selectable. See [Table 10-9](#).
 - MMCR0[PMC2SEL]—PMC2 input selector, 64 events selectable. See [Table 10-10](#).
 - MMCR1[PMC3SEL]—PMC3 input selector, 32 events selectable. See [Table 10-11](#).

- MMCR1[PMC4SEL]—PMC4 input selector, 32 events selectable. See [Table 10-12](#).
- MMCR1[PMC5SEL]—PMC5 input selector, 32 events selectable. See [Table 10-13](#).
- MMCR1[PMC6SEL]—PMC6 input selector, 64 events selectable. See [Table 10-14](#).
- In [Table 10-9](#) through [Table 10-14](#), a correlation is established between each counter, events to be traced, and the pattern required for the desired selection.
- As shown in [Table 10-9](#) through [Table 10-12](#), the first five events are common to all six counters and are considered to be reference events. These are as follows:
 - 00000—Register holds current value
 - 00001—Number of processor cycles
 - 00010—Number of completed instructions, not including folded branches
 - 00011—Number of times the TBL bit transitions from 0 to 1. The TBL bit is specified through MMCR0[TBSEL]
 - 00 = uses the TBL[31] bit to count
 - 01 = uses the TBL[23] bit to count
 - 10 = uses the TBL[19] bit to count
 - 11 = uses the TBL[15] bit to count
 - 00100—Number of instructions dispatched: 0, 1, 2, or 3 per cycle

10.5.1 PMC1 Events

The event to be monitored can be chosen by setting MMCR0[PMC1SEL]. The selected events are counted beginning when MMCR0 is set until either MMCR0 is reset or a performance monitor interrupt is generated. [Table 10-9](#) lists the selectable events and their encodings.

Table 10-9. PMC1 Events—MMCR0[PMC1SEL] Select Encodings

Number	Event	Description
0 (000_0000)	Nothing	Register counter holds current value
1 (000_0001)	Processor cycles	Counts every processor cycle
2 (000_0010)	Instructions completed	Counts all completed PowerPC and AltiVec instructions. Load/store multiple instructions (l_{mw} , st_{mw}) and load/store string instructions (l_{swl} , l_{swx} , st_{swl} , st_{swx}) are only counted once. Does not include folded branches. The counter can increment by 0, 1, 2, or 3, depending on the number of completed instructions per cycle. Branch folding must be disabled (HID0[FOLD] = 0) in order to count all the instructions.
3 (000_0011)	TBL bit transitions	Counts transitions from 0 to 1 of TBL bits specified through MMCR0[TBSEL]. <ul style="list-style-type: none"> 00 = uses the TBL[31] bit to count 01 = uses the TBL[23] bit to count 10 = uses the TBL[19] bit to count 11 = uses the TBL[15] bit to count

Table 10-9. PMC1 Events—MMCR0[PMC1SEL] Select Encodings (continued)

Number	Event	Description
4 (000_0100)	Instructions dispatched	Counts dispatched instructions. The counter can increment by 0, 1, 2, or 3, depending on the number of dispatched instructions per cycle. Load/store multiple instructions (lmw , stmw) and load/store string instructions (lswl , lswx , stswl , stswx) are only counted once. This event includes instructions dispatched directly to the completion queue,
5 (000_0101)	Processor performance monitor exception condition	Counts times the processor begins to generate its performance monitor exception condition. The exception condition is set when the processor performance monitor counter is negative and its interrupt signaling is enabled via MMCR0[PMC1CE] or MMCR0[PMCnCE]. The e600 core does not require MMCR0[PMXE] to be set for the interrupt to occur.
6 (000_0110)	—	Reserved
7 (000_0111)	External performance monitor signal	Counts times the external performance monitor signal ($\overline{pmon_in}$) transitions from negated to asserted.
8 (000_1000)	VPU instructions completed	Counts VPU instructions completed.
9 (000_1001)	VFPU instructions completed	Counts VFPU instruction completed.
10 (000_1010)	VIU1 instructions completed	Counts VIU1 instructions completed.
11 (000_1011)	VIU2 instructions completed	Counts VIU2 instructions completed.
12 (000_1100)	mtvscr instructions completed	Counts completed AltiVec mtvscr instructions.
13 (000_1101)	mtvrsave instructions completed	Counts completed AltiVec mtvrsave instructions.
14 (000_1110)	Cycles a VPU instruction	Counts the cycles an AltiVec instruction in the VPU reservation station is waiting for an operand.
15 (000_1111)	Cycles a VFPU instruction	Counts the cycles an AltiVec instruction in the VFPU reservation station is waiting for an operand.
16 (001_0000)	Cycles a VIU1 instruction	Counts the cycles an AltiVec instruction in the VIU1 reservation station is waiting for an operand.
17 (001_0001)	Cycles an instruction in VIU2 reservation station waits for operand	Counts the cycles an AltiVec instruction in the VIU2 reservation station is waiting for an operand.
18 (001_0010)	mfvschr synchronization	Counts cycles when the VFPU has a valid mfvschr instruction issued, but the instruction cannot start execution because it is not at the bottom of the CQ.
19 (001_0011)	VSCR[SAT] set	Counts whenever VSCR[SAT] goes from 0 to 1.
20 (001_0100)	Store instructions	Counts completed store instructions. Store string and multiples count as single instructions. This count does not include the following instructions, which do not perform a load or store: sync , eieio , dcfb , dcbi , dcbst , dcbt , dcbst , dcbz , icbi , tlbie , tlbid , tlbli , tlbsync , dcba , dst , dstt , dstst , dststt , dss , and dssall .
21 (001_0101)	L1 instruction cache misses	Counts L1 instruction cache misses. This does not include cache-inhibited or cache-disabled accesses.
22 (001_0110)	L1 data snoops	Counts the snoop accesses to the L1 data cache.

Table 10-9. PMC1 Events—MMCR0[PMC1SEL] Select Encodings (continued)

Number	Event	Description
23 (001_0111)	Unresolved branches	Counts branches that are unresolved when processed. This includes branches in a speculative path that might later be thrown away due to another previously predicted branch that mispredicts.
24 (001_1000)	Cycles first speculation buffer active	Counts cycles that a predicted branch is active in the first speculation buffer.
25 (001_1001)	Branch unit stall	Counts cycles the branch unit cannot process new branches. This includes waiting on speculative branches that are not resolved.
26 (001_1010)	True branch target instruction hits	Counts the true branch target instruction hits for taken branches. Note that this count includes speculative branches that have been taken.
27 (001_1011)	Branch link stack predicted	Counts the branches that use link stack prediction. This count includes branches that are in speculative paths. This count may be greater than the sum of link-stack-correctly-resolved and link-stack-mispredicted because another branch may mispredict and cause this branch to be thrown off the link stack before the resolution occurs.
28 (001_1100)	Dispatches to GPR issue queue	Counts instructions dispatched to the GPR issue queue. This includes instructions of speculative paths. Instructions that are executed by the IUs or LSU are dispatched to the GPR issue queue.
29 (001_1.101)	Cycles where 3 instructions are dispatched	Counts cycles where three instructions are dispatched from the dispatch unit.
30 (001_1110)	Counts instruction queue entries over MMCR0[THRESHOLD]	Counts the cycles when the number of valid instruction queue entries is greater than or equal to the MMCR0[THRESHOLD] value. This event does not scale the MMCR0[THRESHOLD] value.
31 (001_1111)	Counts AltiVec issue queue entries over MMCR0[THRESHOLD]	Counts the cycles when the number of valid AltiVec issue queue entries is greater than or equal to the MMCR0[THRESHOLD] value. This event does not scale the MMCR0[THRESHOLD] value.
32 (010_0000)	Cycles where no instructions completed	Counts the cycles where no instructions are completed.
33 (010_0001)	Completed IU2 instructions	Counts IU2 instructions completed.
34 (010_0010)	Completed branch instructions	Counts branches completed, but it does not include folded branches. To count all branches, branch folding must be disabled by clearing H1D0[FOLD].
35 (010_0011)	eieio instructions completed	Counts completed eieio instructions.
36 (010_0100)	mtspr instructions completed	Counts completed mtspr instructions. This count does not include mtvscr instructions.
37 (010_0101)	sc instructions completed	Counts completed system call (sc) instructions.
38 (010_0110)	Load string and load multiple instructions completed	Counts completed load string (lswl , lswx) and load multiple (lm) instructions. Load strings and load multiples are only counted once regardless of how many pieces they are broken into. An lswx instruction of length zero is counted once if MSR[SE] is set; otherwise, it is not counted.

Table 10-9. PMC1 Events—MMCR0[PMC1SEL] Select Encodings (continued)

Number	Event	Description
39 (010_0111)	ITLB hardware table search cycles	Counts cycles spent performing a hardware table search operation for the instruction TLB. A hardware table search begins when the ITLB determines that it has missed and all instructions ahead of the ITLB miss have completed. A hardware table search ends when the page table entry (PTE) or a page fault signal is returned by the table search. Note that the cycles do not include the time it takes the e600 core to drain before the hardware table search operation begins.
40 (010_1000)	DTLB hardware table search cycles over MMCR0[THRESHOLD] value	Counts cycles beyond MMCR0[THRESHOLD] value that are required to perform a hardware data TLB search operation for a data access. This includes table search operations that do not find a matching PTE entry in the page table. This also includes table search operations caused by dst , dstt , dstst , and dststt instructions. This event scales the MMCR0[THRESHOLD] value as specified by MMCR2[THRESHMULT].
41 (010_1001)	L1 instruction cache accesses	Counts the L1 instruction cache accesses. This does not include cache-inhibited or cache-disabled accesses.
42 (010_1010)	Instruction breakpoint matches	Counts when the address of an instruction being completed matches the instruction address breakpoint register (IABR). A match is determined by the following equation: $\text{Match} = ((\text{IABR}[0-29] \& \text{BAMR}[0-29]) == (\text{completion_address}[0-29] \& \text{BAMR}[0-29]))$ The counter can increment by 0, 1, 2, or 3 depending on the number of completed instructions per cycle
43 (010_1011)	L1 data cache load miss cycles over MMCR0[THRESHOLD] value	Counts the cycles an L1 data cache load miss requires beyond the MMCR0[THRESHOLD] value to write back to the rename registers. The cycle count compared to the threshold value represents the number of cycles starting with allocation in the L1 Miss Queue. This event scales the MMCR0[THRESHOLD] value as specified by MMCR2[THRESHMULT]. The miss latency threshold is measured per dispatched operation, not per instruction. Load strings and multiples may have multiple dispatches per instruction. Misaligned loads have multiple accesses, but only one dispatch. The measurement for a misaligned load is from the first piece that misses until the entire load finishes. Note that only the oldest entry of the LMQ is counted.
44 (010_1100)	L1 data snoop hit on modified	Counts snoop accesses to the L1 data cache that hit in a modified cache line.
45 (010_1101)	Load miss alias	Counts loads that alias against an entry already in the L1 miss queue and stalled.
46 (010_1110)	Load miss alias on touch	Counts loads that alias against a touch in the L1 miss queue.
47 (010_1111)	Touch alias	Counts touches that alias against an entry already in the L1 miss queue.
48 (011_0000)	L1 data snoop hit in L1 castout queue	Counts snoop accesses to the L1 data cache that hit in the L1 castout queue and create a push.
49 (011_0001)	L1 data snoop hit castout	Counts snoop accesses to the L1 data cache that hit in a castout and were retried (pre-L1 castout queue or bottom of L1 castout queue).
50 (11_0010)	L1 data snoop hits	Counts snoop accesses to the L1 data cache that hit regardless of the cache state (shared, exclusive, or modified).

Table 10-9. PMC1 Events—MMCR0[PMC1SEL] Select Encodings (continued)

Number	Event	Description
51 (11_0011)	Write-through stores	Counts write-through stores sent to the core memory subsystem after gathering.
52 (11_0100)	Cache-inhibited stores	Counts cache-inhibited stores sent to the core memory subsystem after gathering.
53 (11_0101)	L1 data load hit	Counts each L1 data cache load hit. It does not include MMU table search lookups or touches.
54 (11_0110)	L1 data touch hit	Counts once per dcbt or dcbtst instruction or dstx cache line fetch that hits in the L1 data cache.
55 (11_0111)	L1 data store hit	Counts write-back store hit attempts that successfully hit in the L1 data cache. Does not count if a store hits on a shared cache line. A gathered stores is considered a hit.
56 (11_1000)	L1 data total hits	Counts L1 data cache load, store, or touch hits.
57 (11_1001)	dst instructions dispatched	Counts dst instructions dispatched to VTQ. Includes speculative dst instructions that are canceled.
58 (11_1010)	Refreshed dsts	Counts dst operations issued to already active streams.
59 (11_1011)	Successful dst , dstt , dstst , and dststt table search operations	Counts non-faulting table search operations caused by data stream touch instructions (dst , dstt , dstst , and dststt).
60 (11_1100)	dss instructions completed	Counts dss instructions completed.
61(011_1101)	dst stream 0 cache line fetches	Counts dst stream 0 cache line fetches from the data stream engine (VT0) within the vector-touch queue (VTQ). This includes accesses that hit or miss in the L1 data cache.
62 (011_1110)	VTQ suspends due to change of context	Counts any number of VTQ streams that pause due to a change in MSR[PR] or MSR[DR].
63 (011_1111)	VTQ line fetch hit	Number of VTQ generated accesses that hit in the L1 data cache.
64 (100_0000)	AltiVec load instructions completed	Counts completed AltiVec load instructions.
65 (100_0001)	Floating-point store instructions completed in LSU	Counts aligned floating-point store instructions completed. All misaligned floating-point store instructions completed are counted under PMC1, event number 88 (101_1000).
66 (100_0010)	Floating-point renormalization	Counts times a floating-point store single requires renormalization.
67 (100_0011)	Floating-point denormalization	Counts times a floating-point store double requires denormalization.
68 (100_0100)	Floating-Point store causes stall in LSU	Counts cycles a floating-point store in the FSQ results in a store not being able to complete.
69 (100_0101)	—	Reserved
70 (100_0110)	Load/store true alias stall	Counts times the load or store is stalled due to a true alias.
71 (100_0111)	LSU indexed alias stall	Counts times the LSU RA latch is stalled due to a true or indexed alias.
72 (100_1000)	LSU alias versus FSQ/WB0/WB1	Counts times an alias occurs in LSU for a load versus the finished store queue or write-back stages.
73 (100_1001)	LSU alias versus CSQ	Counts times alias occurs in LSU for a load versus the completed store queue (CSQ).

Table 10-9. PMC1 Events—MMCR0[PMC1SEL] Select Encodings (continued)

Number	Event	Description
74 (100_1010)	LSU load-hit line alias versus CSQ0	Counts times line alias occurs in LSU for a load hit versus a completed store in CSQ0.
75 (100_1011)	LSU load-miss line alias versus CSQ0	Counts times line alias occurs in LSU for a load miss versus completed store in CSQ0.
76 (100_1100)	LSU touch alias versus FSQ/WB0/WB1	Counts times alias occurs in LSU for a touch versus the finished store queue or write-back stages.
77 (100_1101)	LSU touch alias versus CSQ	Counts times alias occurs in the LSU for a touch versus the completed store queue.
78 (100_1110)	LSU LMQ full stall	Counts times the LSU RA latch is stalled while L1 miss queue (LMQ) is full.
79 (100_1111)	Floating-point load instruction completed in LSU	Counts times a floating-point load instruction is completed in the LSU.
80 (101_0000)	Floating-point load single instruction completed in LSU	Counts times a floating-point load single instruction is completed in the LSU.
81 (101_0001)	Floating-point load double completed in LSU	Counts times a floating-point load double instruction is finished in the LSU.
82 (101_0010)	LSU RA latch stall	Counts times the LSU RA latch is stalled for any reason.
83 (101_0011)	LSU load versus store queue alias stall	Counts times an LSU stall exists due to a true alias between a load or touch and the store queue (both Finished store queue(FSQ) and completed store queue (CSQ)).
84 (101_0100)	LSU LMQ index alias	Counts times an LSU stall exists due to an index alias against the LMQ.
85 (101_0101)	LSU store queue index alias	Counts times an LSU stall exists due to an index alias against the store queue.
86 (101_0110)	LSU CSQ forwarding	Counts times the completed store queue forwards to a load in the LSU.
87 (101_0111)	LSU misalign load finish	Counts times a misaligned load finishes in the LSU.
88 (101_1000)	LSU misalign store complete	Counts times a misaligned store completes in the LSU.
89 (101_1001)	LSU misalign stall	Counts times reservation station 0 (RS0) is misaligned and pending a misalign stall.
90 (101_1010)	Floating-point 1/4 FPSCR renames busy	Counts times the FPSCR rename is 1/4 busy.
91 (101_1011)	Floating-point 1/2 FPSCR renames busy	Counts times the FPSCR rename is 1/2 busy.
92 (101_1100)	Floating-point 3/4 FPSCR renames busy	Counts times the FPSCR rename is 3/4 busy.
93 (101_1101)	Floating-point all FPSCR renames busy	Counts times the FPSCR renames are completely busy.
94 (101_1110)	Floating-point denormalized result	Counts times when a floating-point calculation results in a denormalized result.
95-127	—	Reserved

10.5.2 PMC2 Events

MMCR0[PMC2SEL] specify the events associated with PMC2, as shown in [Table 10-10](#).

Table 10-10. PMC2 Events—MMCR0[PMC2SEL] Select Encodings

Number	Event	Description
0 (00_0000)	Nothing	Register counter holds current value.
1 (00_0001)	Processor cycles	Counts every processor cycle.
2 (00_0010)	Instructions completed	Counts all completed PowerPC and Altivec instructions. Load/store multiple instructions (lmw , stmw) and load/store string instructions (lswl , lswx , stswl , stswx) are counted only once. Does not include folded branches. The counter can increment by 0, 1, 2, or 3, depending on the number of completed instructions per cycle. Branch folding must be disabled (HID0[FOLD] = 0) in order to count all the instructions.
3 (00_0011)	TBL bit transitions	Counts transitions from 0 to 1 of TBL bits specified through MMCR0[TBSEL] 00 = uses the TBL[31] bit to count 01 = uses the TBL[23] bit to count 10 = uses the TBL[19] bit to count 11 = uses the TBL[15] bit to count
4 (00_0100)	Instructions dispatched	Counts dispatched instructions. The counter can increment by 0, 1, 2, or 3, depending on the number of completed instructions per cycle. Load/store multiple instructions (lmw , stmw) and load/store string instructions (lswl , lswx , stswl , stswx) are counted only once. This event includes instructions that are dispatched directly to the completion queue.
5 (00_0101)	Processor performance monitor exception	Counts the times the processor begins to generate its performance monitor exception condition. The condition is set when the processor performance monitor counter is negative and interrupt signaling is enabled via MMCR0[PMC1CE] or MMCR0[PMCnCE]. The e600 core does not require MMCR0[PMXE] to be set for the interrupt to occur.
6 (00_0110)	—	Reserved. Read as zero.
7 (00_0111)	External performance monitor signal	Counts times the external performance monitor signal ($\overline{pmon_in}$) transitions from negated to asserted.
8 (00_1000)	VPU instructions completed	Counts VPU instructions completed.
9 (00_1001)	VFPU instructions completed	Counts VFPU instructions completed.
10 (00_1010)	VIU1 instructions completed	Counts VIU1 instructions completed.
11 (00_1011)	VIU2 instructions completed	Counts VIU2 instructions completed.
12 (00_1100)	mtvscr instructions completed	Counts completed mtvscr instructions.
13 (00_1101)	mtvrsave instructions completed	Counts completed mtvrsave instructions.

Table 10-10. PMC2 Events—MMCR0[PMC2SEL] Select Encodings (continued)

Number	Event	Description
14 (00_1110)	Cycles a VPU instruction in the reservation station is waiting for an operand	Counts cycles an AltiVec instruction in the vector permute unit reservation station is waiting for an operand.
15 (00_1111)	Cycles a VFPU instruction in the reservation station is waiting for operand	Counts the cycles an AltiVec instruction in the VFPU reservation station is waiting for an operand.
16 (01_0000)	Cycles a VIU1 instruction in the reservation station is waiting for operand	Counts the cycles an AltiVec instruction in the VIU1 reservation station is waiting for an operand.
17 (01_0001)	Cycles a VIU2 instruction in the reservation station is waiting for operand	Counts the cycles an AltiVec instruction in the VIU2 reservation station is waiting for an operand.
18 (01_0010)	mfvscr synchronization	Counts cycles when the VFPU has a valid mfvscr instruction dispatched, but the instruction cannot be completed because it is not at the bottom of the completion queue.
19 (01_0011)	VSCR[SAT] set	Counts whenever VSCR[SAT] goes from 0 to 1.
20 (01_0100)	Store instructions	Counts store instructions completed. Store strings and store multiples count only once. This count does not include instructions like cache operations that do not actually perform a load or store. Instructions not counted in the store instructions event are: sync, eieio, dcbf, dcbl, dcbst, dcbl, dcbtst, dcbz, icbl, tlbie, tlbld, tlbli, tlbsync, dcba, dst, dstt, dstst, dststt, dss, and dssall .
21 (01_0101)	L1 instruction cache misses	Counts the L1 instruction cache misses. This does not include cache inhibited or cache-disabled accesses.
22 (01_0110)	L1 data snoops	Counts snoop accesses to the L1 data cache.
23 (01_0111)	L1 data total misses	Counts L1 data cache load, store, or touch misses.
24 (01_1000)	Dispatches to FPR issue queue	Counts the instructions dispatched to the FPR issue queue. This includes instructions in the speculative paths. Instructions executed by the FPU are dispatched to the FPR issue queue.
25 (01_1001)	LSU instructions completed	Counts LSU instructions completed
26 (01_1010)	Load instructions	Counts the load instructions completed. Load strings and load multiples only count once. This count does not include instructions like cache operations that do not actually perform a load or store. Instructions not counted in the load instructions event are: sync, eieio, dcbf, dcbl, dcbst, dcbl, dcbtst, dcbz, icbl, tlbie, tlbld, tlbli, tlbsync, dcba, dst, dstt, dstst, dststt, dss, dssall, and lswx with length zero (XER[25–31] = 0).
27 (01_1011)	Store string and store multiple instructions	Counts store string and store multiple instructions completed. Store strings and store multiples are counted only once. A stswx instruction with length zero (XER[25–31] = 0) is counted once.

Table 10-10. PMC2 Events—MMCR0[PMC2SEL] Select Encodings (continued)

Number	Event	Description
28 (01_1100)	tlbie instructions completed	Counts tlbie instructions completed.
29 (01_1101)	lwarx instructions completed	Counts lwarx instructions completed.
30 (01_1110)	mf spr instructions completed	Counts mf spr instructions completed. This count does not include mf vscr instructions.
31 (01_1111)	Refetch serialization	Counts when a refetch serialization occurs for the following cases: <ul style="list-style-type: none"> • isync completes • sc completes • rfi completes • When an instruction that sets the XER[SO] bit completes (Changes XER[SO] from zero to one.) and when XER[SO] is cleared by a mf spr instruction • Interrupts taken • Tracing is enabled (MSR[SE] = 1 or MSR[BE] = 1) and a branch is speculative at branch processing time • Floating-point exception cases where the <i>Programming Environments Manual</i> specifies that the target FPR is unchanged. • dcba to a page marked as write-through or cache-inhibited.
32 (10_0000)	Completion queue entries over MMCR0[THRESHOLD] value	Counts the cycles when the valid completion queue entries are greater than or equal to the MMCR0[THRESHOLD] value. This event does not scale the MMCR0[THRESHOLD] value.
33 (10_0001)	Completing one instruction	Counts cycles in which exactly 1 instruction is completed.
34 (10_0010)	Two instructions dispatched	Counts cycles in which exactly 2 instructions are dispatched.
35 (10_0011)	ITLB non-speculative misses	Counts times that a requested non-speculative address translation was not in the instruction TLB.
36 (10_0100)	Cycles waiting from L1 instruction cache miss	Counts cycles spent waiting for L1 instruction cache miss. This includes all instruction fetches, both cacheable and cache-inhibited. It counts from when the miss is detected until either the data is returned or the request is cancelled.
37 (10_0101)	L1 data load access miss	Counts L1 data cache load access misses. This does not include MMU table search lookups or touches.
38 (10_0110)	L1 data touch miss	Counts once for every dstx cache line fetch, dcbt , or dcbtst L1 data cache miss that causes an L1 data cache reload
39 (10_0111)	L1 data store miss	Counts write-back store attempts that missed in the L1 data cache. This counts only once on gathered stores, if gathered before the cache access.
40 (10_1000)	L1 data touch miss cycles	Counts cycles spent waiting for L1 data cache touch misses from when the miss is detected until either the data is returned or the request is cancelled.
41 (10_1001)	L1 data cycles used	Counts cycles when the L1 data cache is used for any reason but does not include snoop accesses. The count value indicates the L1 data cache bandwidth consumed when compared to the number of processor cycles elapsed.

Table 10-10. PMC2 Events—MMCR0[PMC2SEL] Select Encodings (continued)

Number	Event	Description
42 (10_1010)	dst stream 1 cache line fetches	Counts dst stream 1 cache line fetches from the data stream engine (VT1) within the vector touch queue (VTQ). This includes accesses that hit or miss in the L1 data cache.
43 (10_1011)	VTQ stream cancelled prematurely	Counts times when a VTQ stream is cancelled due to branch speculation cancel, inappropriate translation protection, or WIMG. This does not include cases where the VTQ stream is cancelled by reaching the end of a stream, refresh, or dss . This counter can increment by 0, 1, 2, 3 or 4 at a time.
44 (10_1100)	VTQ resumes due to change of context	Counts any time the VTQ streams resume due to change in MSR[PR] or MSR[DR].
45 (10_1101)	VTQ line fetch miss	Counts VTQ generated accesses that miss in the L1 data cache.
46 (10_1110)	VTQ line fetch	Counts all VTQ fetch attempts. This includes all VTQ generated accesses that hit or miss in the L1 data cache.
47 (10_1111)	TLBIE snoops	Counts the TLB invalidations performed due to another master's TLBIE broadcast.
48 (11_0000)	L1 instruction cache reloads	Counts times that the L1 instruction cache is reloaded with a new cache line. This does not include cache-inhibited accesses. It does include accesses with the cache disabled and accesses that miss in the data cache when all ways are locked.
49 (11_0001)	L1 data cache reloads	Counts times that the L1 data cache is reloaded with a new cache line. This does not include cache-inhibited accesses, accesses with the cache disabled, or accesses that miss in the data cache when all ways are locked.
50 (011_0010)	L1 data cache castouts to L2	Counts L1 data cache castouts to the L2 cache.
51 (011_0011)	Store merge/gather	Counts store operations that are merged with other store operations in the completed store queue (CSQ).
52 (011_0100)	Cacheable store merge to 32 bytes	Counts times all 32 bytes have been merged in the completed store queue (CSQ) to allow a full cache line write operation.
53 (011_0101)	Data breakpoint matches	Counts times a data address breakpoint interrupt is signalled.
54 (011_0110)	Fall-through branches processed	Counts branches that were either predicted or resolved as not-taken. This includes branches that are in a speculative path that might later be thrown away due to another previously predicted branch that mispredicts.
55 (011_0111)	First speculative branch buffer resolved correctly	Counts branches in the first prediction buffer that resolve correctly. out of order branch resolution means that some parts of this count may be due to branches in a speculative path that resolve correctly, but the speculative path is later mispredicted.
56 (011_1000)	Second speculation buffer active	Counts the cycles that a predicted branch is active in the second speculation buffer.
57 (011_1001)	BPU Stall on LR dependency	Counts the cycles the branch processing unit (BPU) stalls due to the link register (LR) being unresolved. If the link stack is enabled, a stall on LR dependency occurs only when the LR is unavailable and the link stack is empty. The count includes stalls down speculative paths.
58 (011_1010)	BTIC miss	Counts branch target instruction cache (BTIC) misses for taken branches. Note that this count includes taken branches that are in speculative paths.
59 (011_1011)	Branch link stack correctly resolved	Counts branches that use link stack prediction and resolve correctly. This count includes branches that are in speculative paths.

Table 10-10. PMC2 Events—MMCR0[PMC2SEL] Select Encodings (continued)

Number	Event	Description
60 (011_1100)	FPR issue stalled	Counts times an instruction in the FPR issue queue could not be issued. This should only occur if the FPU is busy when an instruction is ready to issue.
61(011_1101)	Switches between Privileged and User	Counts times the MSR[PR] bit gets set and cleared.
62 (011_1110)	LSU completes floating-point store single	Counts aligned floating-point store single instructions completed. All misaligned floating-point store instructions completed are counted under PMC1, event number 88 (101_1000).
63 (011_1111)	—	Reserved

10.5.3 PMC3 Events

Bits MMCR1[PMC3SEL] specify events associated with PMC3, as shown in [Table 10-11](#).

Table 10-11. PMC3 Events—MMCR1[PMC3SEL] Select Encodings

Number	Event	Description
0 (0_0000)	Nothing	Register counter holds current value.
1 (0_0001)	Processor cycles	Counts every processor clock cycle.
2 (0_0010)	Instructions completed	Counts all completed PowerPC and Altivec instructions. Load/store multiple instructions (lmw , stmw) and load/store string instructions (lswl , lswx , stswl , stswx) are only counted once. Does not include folded branches. The counter can increment by 0, 1, 2, or 3, depending on the number of completed instructions per cycle. Branch folding must be disabled (HID0[FOLD] = 0) in order to count all the instructions.
3 (0_0011)	TBL bit transitions	Counts transitions from 0 to 1 of TBL bits specified through MMCR0[TBSEL] 00 = uses the TBL[31] bit to count 01 = uses the TBL[23] bit to count 10 = uses the TBL[19] bit to count 11 = uses the TBL[15] bit to count
4 (0_0100)	Instructions dispatched	Counts dispatched instructions. The counter can increment by 0, 1, 2, or 3, depending on the number of completed instructions per cycle. Load/store multiple instructions (lmw , stmw) and load/store string instructions (lswl , lswx , stswl , stswx) are counted only once. This event includes instructions that are dispatched directly to the completion queue.
5 (0_0101)	Processor performance monitor exception	Counts times the processor begins to generate its performance monitor exception condition. The exception condition is set when the processor performance monitor counter is negative and interrupt signaling is enabled via MMCR0[PMC1CE] or MMCR0[PMCnCE]. The e600 core does not require MMCR0[PMXE] to be set to allow the interrupt to occur.
6 (0_0110)	—	Reserved Read as zero.
7 (0_0111)	External performance monitor signal	Counts times the external performance monitor signal ($\overline{pmon_in}$) transitions from negated to asserted.
8 (0_1000)	Completing two instruction	Counts cycles when exactly two instruction are completed.

Table 10-11. PMC3 Events—MMCR1[PMC3SEL] Select Encodings (continued)

Number	Event	Description
9 (0_1001)	One instruction dispatched	Counts cycles when exactly one instruction is dispatched.
10 (0_1010)	Dispatches to VR issue queue	Counts the instructions dispatched to the vector register (VR) issue queue. This includes instructions in speculative paths. AltiVec instructions are executed by the VPU, VIU1, VIU2, and VFPU are dispatched to the VR issue queue.
11 (0_1011)	VR Stalls	Counts when an instruction in the vector register (VR) issue queue could not be issued. This counter can be incremented by 0, 1, or 2. An AltiVec instruction cannot be issued when its vector execution unit is busy or the AltiVec instruction ahead in the AltiVec issue queue could not be issued.
12 (0_1100)	GPR rename buffer entries over MMCR0[THRESHOLD]	Counts the cycles when the number of valid GPR rename buffers is greater than or equal to the MMCR0[THRESHOLD] value. This event does not scale the MMCR0[THRESHOLD] value.
13 (0_1101)	FPR issue queue entries	Counts the number of valid FPR issue queue entries each cycle.
14 (0_1110)	FPU instructions	Counts FPU instructions completed.
15 (0_1111)	stwcx. instructions	Counts stwcx. instructions completed.
16 (1_0000)	Load string and multiple instruction pieces	Counts pieces of load string and load multiple instructions that are completed. An lswx instruction of length zero is counted once if MSR[SE] is set; otherwise it is not counted.
17 (1_0001)	ITLB hardware table search cycles over threshold	Counts times an instruction TBL hardware search operation for an instruction fetch requires more than the threshold number of cycles to complete. This includes table search operations that do not find any matching PTE entry in the page table. This event scales the MMCR0 threshold value as specified by MMCR2[THRESHMULT].
18 (1_0010)	DTLB misses	Counts times a needed non-speculative data address translation was not in the DTLB.
19 (1_0011)	Cancelled L1 instruction cache misses	Counts cacheable instruction accesses that miss in the instruction cache, but are cancelled before they are accepted by the core memory subsystem.
20 (1_0100)	L1 data cache operation hit	Counts cache operations that hit in the L1 data cache (dcbf, dcbst).
21 (1_0101)	L1 data load miss cycles	Counts cycles spent waiting for L1 data cache misses in the LMQ. It counts from when the miss is detected until either the data is returned or the request is cancelled. Misses in the LMQ include all load and touch operations. Note that a load miss is only counted if it is the oldest entry of the LMQ.
22 (1_0110)	L1 data Pushes	Counts L1 data pushes caused by snoops to modified cache lines.
23 (1_0111)	L1 data total miss	Counts L1 data cache load, store, or touch misses.
24 (1_1000)	VT2 fetches	Counts fetch attempts from the data stream engine 2 (VT2) within the vector-touch queue (VTQ). This includes accesses that hit or miss in the L1 data cache.
25 (1_1001)	Taken branches that are processed	Counts branches that were either predicted or resolved taken. This includes branches that are in a speculative path. This also includes branches that are in a speculative path that might later be thrown away due to another previously predicted branch that mispredicts.
26 (1_1010)	Branch flushes	Counts flushes for clearing mispredicted instructions out of the completion queue.

Table 10-11. PMC3 Events—MMCR1[PMC3SEL] Select Encodings (continued)

Number	Event	Description
27 (1_1011)	Second speculative branch buffer resolved correctly	Counts branches in the second prediction buffer that resolve correctly. Out of order branch resolution means that some parts of this count may be due to branches in a speculative path that resolve correctly, but the speculative path is later mispredicted.
28 (1_1100)	Third speculation buffer active	Counts cycles that a third predicted branch is active.
29 (1_1101)	Branch unit stall on CTR dependency	Counts cycles the branch unit is stalled due to the counter register (CTR) being unresolved. Includes stalls down speculative paths.
30 (1_1110)	Fast BTIC hit	Counts FBTIC hits for taken branches. This number should be greater than or equal to the BTIC hit count. The difference between this count and the BTIC hit count provides the number of aliased BTIC hits. Aliased BTIC hits force a hiccup in the fetch pipe, delaying when the instructions at the branch target address are available for dispatch. Note that this count includes taken branches that are in speculative paths.
31 (1_1111)	Branch link stack mispredicted	Counts branches that use Link Stack Prediction and resolve incorrectly. This count includes branches that are in speculative paths.

10.5.4 PMC4 Events

Bits MMCR1PMC4SEL] specify events associated with PMC4, as shown in [Table 10-12](#).

Table 10-12. PMC4 Events—MMCR1[PMC4SEL] Select Encodings

Number	Event	Description
0 (0_0000)	Nothing	Register counter holds current value.
1 (0_0001)	Processor cycles	Counts every processor cycle.
2 (0_0010)	Instructions completed	Counts completed PowerPC and Altivec instructions. Load/store multiple/string instructions are only counted once even though they are broken up into pieces. Does not include folded branches. To count all instructions, HID0[FOLD] must be cleared to disable branch folding.
3 (0_0011)	TBL bit transitions	Counts transitions from 0 to 1 of TBL bits specified through MMCR0[TBSEL] 00 = uses the TBL[31] bit to count 01 = uses the TBL[23] bit to count 10 = uses the TBL[19] bit to count 11 = uses the TBL[15] bit to count
4 (0_0100)	Instructions dispatched	Counts dispatched instructions. The counter can increment by 0, 1, 2, or 3, depending on the number of completed instructions per cycle. Load/store multiple instructions (l_{mw} , st_{mw}) and load/store string instructions (l_{swl} , l_{swx} , st_{swl} , st_{swx}) are only counted once. This event includes instructions that are dispatched directly to the completion queue.
5 (0_0101)	Processor performance monitor exception	Counts the times the processor begins to generate its performance monitor exception condition. The exception condition is set when the processor performance monitor counter is negative and its interrupt signaling is enabled via MMCR0[PMC1CE] or MMCR0[PMCE]. The e600 core does not require MMCR0[PMXE] to be set for the interrupt to occur.
6 (0_0110)	—	Reserved. Read as zero.

Table 10-12. PMC4 Events—MMCR1[PMC4SEL] Select Encodings (continued)

Number	Event	Description
7 (0_0111)	External performance monitor signal	Counts times the external performance monitor signal ($\overline{pmon_in}$) transitions from negated to asserted.
8 (0_1000)	Instructions completed in VPU	Counts completed VPU instructions.
9 (0_1001)	Instructions completed in VFPU	Counts completed vector VFPU instructions.
10 (0_1010)	VIU1 instructions completed	Counts completed VIU1 instructions.
11 (0_1011)	VIU2 Instructions completed	Counts completed VIU2 instructions.
12 (0_1100)	mtvscr Instructions completed	Counts completed mtvscr instructions.
13 (0_1101)	mtvrsave Instructions completed	Counts completed mtvrsave instructions.
14 (0_1110)	Completing 3 instructions	Counts cycles where three instructions are completed.
15 (0_1111)	Dispatching 0 instructions	Counts cycles where zero instructions are dispatched.
16 (1_0000)	GPR issue queue entries over threshold	Counts cycles when the valid GPR issue queue entries are greater than or equal to the MMCR0[THRESHOLD] value. This event cannot scale the MMCR0[THRESHOLD] value.
17 (1_0001)	GPR issue queue stalled	Counts cycles that instructions in the GPR issue queue are not issued. This value only increments by 1 on any given cycle. A GPR instruction is not issued when its unit is busy, or when an instruction ahead of it in the GPR issue queue could not issue. An IU1 instruction goes to any non-busy IU1 so it only stalls if more IU1 instructions are trying to issue than there are non-busy IU1 units.
18 (1_0010)	IU1 instructions	Counts completed IU1 instructions.
19 (1_0011)	dssall instructions	Counts completed dssall instructions.
20 (1_0100)	tibsync instructions	Counts completed tibsync instructions.
21 (1_0101)	sync instructions	Counts completed sync instructions.
22 (1_0110)	Store string and multiple instruction pieces	Counts completed pieces of store string and store multiple instructions. An stswx instruction of length zero is counted once.
23 (1_0111)	DTLB hardware table search cycles	Counts cycles spent performing hardware table search operations for DTLB misses. A hardware table search begins when the DTLB determines that it has missed and all instructions ahead of the DTLB miss have completed. A hardware table search ends when the page table entry (PTE) or a page fault signal is returned by the table search engine. The number of cycles does NOT include the time it takes the machine to drain before the hardware table search begins.

Table 10-12. PMC4 Events—MMCR1[PMC4SEL] Select Encodings (continued)

Number	Event	Description
24 (1_1000)	Snoop retries	Counts the number of load-store snoops that are retried by the load-store. This includes snoops external to the core that are retried because of a load-store collision, as well as internal load-store self-snoop retries. It does not include snoops which are retried because of an MSS collision or busy condition. An example of an internal self-snoop collision is a load L1 miss which collides with a castout in the L1 castout queue. This type of collision is handled through internal snoop retry instead of load-store pipeline stall.
25 (1_1001)	Successful stwcx .	Counts stwcx . instructions that completed with reservation intact.
26 (1_1010)	dst stream 3 cache line fetches	Counts dst stream 3 cache line fetches from the data stream engine (VT3) within the vector-touch queue (VTQ). This includes accesses that hit or miss in the L1 data cache.
27 (1_1011)	Third speculative branch buffer resolved correctly	Counts branches in the third prediction buffer that resolve correctly. Out-of-order branch resolution means that some parts of this count may be due to branches in a speculative path that resolve correctly, but the speculative path is later mispredicted.
28 (1_1100)	Mispredicted branches	Counts mispredicted branches. Due to out-of-order branch resolution, this count includes mispredicted branches down speculative paths that may later be mispredicted themselves.
29 (1_1101)	Folded branches	Counts branches actually folded in the instruction queue. Note that this count includes branches that are on speculative paths.
30 (1_1110)	Floating-point store double completes in LSU	Counts aligned floating-point store double instructions completed. All misaligned floating-point store instructions completed are counted under PMC1, event number 88 (0x101_1000).
31 (1_1111)	—	Reserved

10.5.5 PMC5 Events

Bits MMCR1[PMC5SEL] specify events associated with PMC5, as shown in [Table 10-13](#).

Table 10-13. PMC5 Events—MMCR1[PMC5SEL] Select Encodings

Number	Event	Description
0 (0_0000)	Nothing	Register counter holds current value.
1 (0_0001)	Processor cycles	Counts every processor clock cycle.
2 (0_0010)	L2 cache hits	Counts accesses from the processor that hit in the L2 for loads, caching-allowed write-back stores, dcbz , instruction fetches, and touches.
3 (0_0011)	—	Reserved
4 (0_0100)	L2 instruction cache misses	Counts instruction accesses from the processor that miss in the L2 cache.
5 (0_0101)	—	Reserved
6 (0_0110)	L2 data cache misses	Counts data accesses from the processor that miss in the L2 cache (loads, caching-allowed write-back stores, dcbz , and touches).
7 (0_0111)	—	Reserved
8 (0_1000)	L2 load hits	Counts load accesses from the processor that hit in the L2 cache.

Table 10-13. PMC5 Events—MMCR1[PMC5SEL] Select Encodings (continued)

Number	Event	Description
9 (0_1001)	L2 store hits	Counts caching-allowed write-back store accesses from the processor that hit in the L2 cache.
10–12	—	Reserved
13 (0_1101)	L2 touch hits	Counts touch accesses (dcbt , dcbtst , and VTQ) from the processor that hit in the L2 cache.
14 (0_1110)	—	Reserved
15 (0_1111)	Snoop retries	Counts the number of internal requests that are internally retried. This includes load-store retries as well as some MSS collision cases (that would prevent an L2 hit from being considered good).
16 (1_0000)	Snoop modified	Counts times a snoop response to an access made by the processor is modified (internal snooping).
17 (1_0001)	Snoop valid	Counts times a snoop response to an access made by the processor is valid (internal snooping).
18 (1_0010)	Intervention	Counts local interventions serviced by the processor (internal snooping).
19 (1_0011)	L2 cache misses	Counts accesses from the processor that miss in the L2 for loads, caching-allowed write-back stores, dcbz , instruction fetches, and touches.
20–31	—	Reserved

10.5.6 PMC6 Events

The event to be monitored can be chosen by setting MMCR1[15–20]. The selected events are counted beginning when MMCR1 is set until either MMCR1 is reset or a performance monitor interrupt is generated. [Table 10-14](#) lists the selectable events and their encodings.

Table 10-14. PMC6 Events—MMCR1[PMC6SEL] Select Encodings

Number	Event	Description
0 (00_0000)	Nothing	Register counter holds current value.
1 (00_0001)	Processor cycles	Counts every processor cycle.
2 (00_0010)	L2 cache hits	Counts accesses from the processor that hit in the L2 for loads, caching-allowed write-back stores, dcbz , instruction fetches, and touches.
3 (0_0011)	—	Reserved
4 (00_0100)	L2 instruction cache misses	Counts instruction accesses from the processor that miss in the L2 cache.
5 (0_0101)	—	Reserved
6 (00_0110)	L2 data cache misses	Counts data accesses from the processor that miss in the L2 cache (loads, caching-allowed write-back stores, dcbz , and touches).
7 (0_0111)	—	Reserved
8 (00_1000)	L2 cache castouts	Counts L2 cache castouts.
9 (0_1001)	—	Reserved

Table 10-14. PMC6 Events—MMCR1[PMC6SEL] Select Encodings (continued)

Number	Event	Description
10 (00_1010)	L2SQ full cycles	Counts cycles the L2 castout queue (L2SQ) is full (not counting the reserved push slot).
11–12	—	Reserved
13 (00_1101)	L2 touch hits	Counts touch accesses (dcbt , dcbtst , and VTQ) from the processor that hit in the L2 cache.
14 (0_1110)	—	Reserved
15 (00_1111)	Snoop retries	Counts times a snoop response to any access is “retry.”
16–17	—	Reserved
18 (01_0010)	Intervention	Counts local interventions serviced by the processor (internal snooping).
19 (01_0011)	L1 external Interventions	Counts L1 external interventions (External snoop hits modified in the L1 data cache).
20 (01_0100)	L2 external Interventions	Counts L2 interventions caused by external snoops to modified blocks.
21 (01_0101)	—	Reserved
22 (01_0110)	External interventions	Counts external interventions serviced. This is the sum of the L1 and L2 external interventions.
23 (01_0111)	External pushes	Counts times an external snoop causes a push or upgraded castout.
24 (01_1000)	External snoop retry	Counts the number of snoops external to the core that get a retry response.
25 (01_1001)	DTQ full cycles	Counts cycles the DTQ is full (not counting reserved push slot).
26 (01_1010)	MPX bus retry	Counts transactions that were initiated by this core that were retried on the core interface.
27 (01_1011)	L2 valid request	Counts requests serviced by the L2 cache.
28 (01_1100)	BORDQ full	Counts cycles the BORDQ (bus outstanding read queue) is full. The entries in BORDQ correspond directly to the addresses of entries in the LLQ.
29 (01_1101)	L2 cache misses	Counts accesses from the processor that miss in the L2 for loads, caching-allowed write-back stores, dcbz , instruction fetches, and touches.
30–41	—	Reserved
42 (10_1010)	MPX bus $\overline{T}A$ s for reads	Counts external $\overline{T}A$ s received on the MPX bus for all read operations initiated by the processor.
43 (10_1011)	MPX bus $\overline{T}A$ s for writes	Counts external $\overline{T}A$ s received on the MPX bus for all write operations initiated by the processor. This includes $\overline{T}A$ s to which the processor is providing intervention data.
44 (10_1100)	MPX bus reads not retried	Counts load-type operations initiated by the processor on the MPX bus that complete with a non-retry response.
45 (10_1101)	MPX bus writes not retried	Counts store-type operations initiated by the processor on the MPX bus that complete with a non-retry response. This event does not include external push operations that are counted in another event.
46 (10_1110)	MPX bus reads/writes not retries	Counts the total load-type, store-type, and external push operations initiated by the processor on the MPX bus that complete with a non-retry response.

Table 10-14. PMC6 Events—MMCR1[PMC6SEL] Select Encodings (continued)

Number	Event	Description
47 (10_1111)	MPX bus retry due to L1 retry	Counts times retry is asserted on the MPX bus due to an internal L1 retry condition.
48 (11_0000)	MPX bus retry due to previous adjacent	Counts times retry is asserted on the MPX bus due to an internal previous adjacent retry condition.
49 (11_0001)	MPX bus retry due to collision	Counts times retry is asserted on the MPX bus due to an internal collision.
50 (11_0010)	MPX bus retry due to intervention ordering	Counts times retry is asserted on the MPX bus due to an intervention ordering condition.
51 (11_0011)	Snoop requests	Counts qualified snoop requests processed by the snooper.
52 (11_0100)	Prefetch engine request	Counts new prefetches allocated in the prefetch unit.
53 (11_0101)	Prefetch engine collision vs. load	Counts times a load collides with an outstanding prefetch request from the L2 prefetch engine while accessing L2.
54 (11_0110)	Prefetch engine collision vs. store	Counts times a store collides with an outstanding prefetch request from the L2 prefetch engine while accessing L2.
55 (11_0111)	Prefetch engine collision vs. i instruction fetch	Counts times an instruction fetch collides against an outstanding request from the L2 prefetch engine while accessing L2.
56 (11_1000)	Prefetch engine collision vs. load/store/instruction fetch	Counts times the L2 prefetch engine collides against an outstanding load, store, or instruction fetch in the load miss queue while accessing L2.
57 (11_1001)	Prefetch engine full	Counts times an L2 prefetch is not initiated because the prefetch engine is full while accessing L2.
58–63	—	Reserved

Appendix A

e600 Core Instruction Set Listings

This appendix lists the e600 core’s instruction set as well as the additional PowerPC instructions not implemented in the e600 core. The instructions are sorted by mnemonic, opcode, function, and form. This appendix also includes a quick reference table that contains general information, such as the architecture level, privilege level, and form, and indicates if the instruction is 64-bit and optional.

NOTES

The e600 core does not implement any 64-bit instructions.

Split fields, that represent the concatenation of sequences from left to right, are shown in lowercase. For more information refer to Chapter 8, “Instruction Set,” in the *Programming Environments Manual*.

A.1 Instructions Sorted by Mnemonic (Decimal and Hexadecimal)

Table A-1 shows the instructions implemented in the e600 core. The instructions are listed in alphabetical order by their mnemonic name. The primary opcode (0–5) and secondary opcode (21–31) are decimal and hexadecimal values.

Key:

 Reserved bits

Table A-1. Instructions by Mnemonic (Dec, Hex)

Name	0	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
addx	31 (0x1F)				D					A					B			OE										Rc
addcx	31 (0x1F)				D					A					B			OE										Rc
addex	31 (0x1F)				D					A					B			OE										Rc
addi	14 (0xE)				D					A					SIMM													
addic	12 (0xC)				D					A					SIMM													
addic.	13 (0xD)				D					A					SIMM													
addis	15 (0xF)				D					A					SIMM													
addmex	31 (0x1F)				D					A					0_0000			OE										Rc
addzex	31 (0x1F)				D					A					0_0000			OE										Rc
andx	31 (0x1F)				S					A					B													Rc
andcx	31 (0x1F)				S					A					B													Rc

Table A-1. Instructions by Mnemonic (Dec, Hex) (continued)

Name	0	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31							
andi.	28 (0x1C)	S			A			UIMM																											
andis.	29 (0x1D)	S			A			UIMM																											
bx	18 (0x12)	LI																								AA	LK								
bcx	16 (0x10)	BO			BI			BD																					AA	LK					
bcctrx	19 (0x13)	BO			BI			0_0000			00528 (0x210)																					LK			
bclrx	19 (0x13)	BO			BI			0_0000			0016 (0x010)																					LK			
cmp	31 (0x1F)	crfD	0	L	A			B			0000 (0x000)																					0			
cmpi	11 (0x0B)	crfD	0	L	A			SIMM																											
cmpl	31 (0x1F)	crfD	0	L	A			B			0032 (0x020)																						0		
cmpli	10 (0x0A)	crfD	0	L	A			UIMM																											
cntlzwx	31 (0x1F)	S			A			0_0000			0026 (0x01A)																					Rc			
crand	19 (0x13)	crbD			crbA			crbB			0257 (0x101)																						0		
crandc	19 (0x13)	crbD			crbA			crbB			0129 (0x081)																						0		
creqv	19 (0x13)	crbD			crbA			crbB			0289 (0x121)																						0		
crnand	19 (0x13)	crbD			crbA			crbB			0225 (0x0E1)																						0		
crnor	19 (0x13)	crbD			crbA			crbB			0033 (0x21)																						0		
cror	19 (0x13)	crbD			crbA			crbB			0449 (0x1C1)																						0		
crorc	19 (0x13)	crbD			crbA			crbB			0417 (0x1A1)																						0		
crxor	19 (0x13)	crbD			crbA			crbB			0193 (0C1)																						0		
dcba ¹	31 (0x1F)	000_00			A			B			0758 (0x2F6)																						0		
dcbf	31 (0x1F)	000_00			A			B			0086 (0x056)																						0		
dcbi ²	31 (0x1F)	000_00			A			B			0470 (0x1D6)																						0		
dcbst	31 (0x1F)	000_00			A			B			0054 (0x036)																						0		
dcbt	31 (0x1F)	000_00			A			B			0278 (0x116)																						0		
dcbtst	31 (0x1F)	000_00			A			B			0246 (0x0F6)																						0		
dcbz	31 (0x1F)	000_00			A			B			1014 (0x3F6)																						0		
divwx	31 (0x1F)	D			A			B			OE	0491 (0x1EB)																				Rc			
divwux	31 (0x1F)	D			A			B			OE	0459 (0x1CB)																					Rc		
dss ³	31 (0x1F)	0	00	STRM	00_000			0_0000			0822 (0x336)																					0			
dssall ³	31 (0x1F)	1	00	STRM	00_000			0_0000			0822 (0x336)																						0		
dst ³	31 (0x1F)	0	00	STRM	A			B			0342 (0x156)																						0		
dstst ³	31 (0x1F)	0	00	STRM	A			B			0374 (0x09C)																						0		
dststt ³	31 (0x1F)	1	00	STRM	A			B			0374 (0x176)																						0		

Table A-1. Instructions by Mnemonic (Dec, Hex) (continued)

Name	0	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
dstt ³	31 (0x1F)	1	00	STRM		A		B		0342 (0x0B0)				0														
eciwx ¹	31 (0x1F)	D				A		B		0310 (0x136)				0														
ecowx ¹	31 (0x1F)	S				A		B		0438 (0x1B6)				0														
eieio	31 (0x1F)	000_00				00_000		0_0000		0854 (0x356)				0														
eqvx	31 (0x1F)	S				A		B		0284 (0x11C)				Rc														
extsbx	31 (0x1F)	S				A		0_0000		0954 (0x3BA)				Rc														
extshx	31 (0x1F)	S				A		0_0000		0922 (0x39A)				Rc														
fabsx	63 (0x3F)	D				00_000		B		0264 (0x108)				Rc														
faddx	63 (0x3F)	D				A		B		0000_0		0021 (0x015)		Rc														
faddsx	59 (0x3B)	D				A		B		0000_0		0021 (0x015)		Rc														
fcmpo	63 (0x3F)	crfD		00		A		B		0032 (0x020)				0														
fcmpu	63 (0x3F)	crfD		00		A		B		0000 (0x000)				0														
fctiw_x	63 (0x3F)	D				00_000		B		0014 (0x00E)				Rc														
fctiwz_x	63 (0x3F)	D				00_000		B		0015 (0x00F)				Rc														
fdiv_x	63 (0x3F)	D				A		B		0000_0		0018 (0x012)		Rc														
fdivs_x	59 (0x3B)	D				A		B		0000_0		0018 (0x012)		Rc														
fmadd_x	63 (0x3F)	D				A		B		C		0029 (0x01D)		Rc														
fmaddsx	59 (0x3B)	D				A		B		C		0029 (0x01D)		Rc														
fmr_x	63 (0x3F)	D				00_000		B		0072 (0x48)				Rc														
fmsub_x	63 (0x3F)	D				A		B		C		0028 (0x01C)		Rc														
fmsubs_x	59 (0x3B)	D				A		B		C		0028 (0x01C)		Rc														
fmul_x	63 (0x3F)	D				A		0_0000		C		0025 (0x019)		Rc														
fmuls_x	59 (0x3B)	D				A		0_0000		C		0025 (0x019)		Rc														
fnabs_x	63 (0x3F)	D				00_000		B		0136 (0x88)				Rc														
fneg_x	63 (0x3F)	D				00_000		B		0040 (0x28)				Rc														
fnmadd_x	63 (0x3F)	D				A		B		C		0031 (0x01F)		Rc														
fnmaddsx	59 (0x3B)	D				A		B		C		0031 (0x01F)		Rc														
fnmsub_x	63 (0x3F)	D				A		B		C		0030 (0x01E)		Rc														
fnmsubs_x	59 (0x3B)	D				A		B		C		0030 (0x01E)		Rc														
fres_x ¹	59 (0x3B)	D				00_000		B		0000_0		0024 (0x018)		Rc														
frsp_x	63 (0x3F)	D				00_000		B		0012 (0xC)				Rc														
frsqrte_x ¹	63 (0x3F)	D				00_000		B		0000_0		0026 (0x01A)		Rc														
fsel_x ¹	63 (0x3F)	D				A		B		C		0023 (0x017)		Rc														

Table A-1. Instructions by Mnemonic (Dec, Hex) (continued)

Name	0	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
fsqrtx ⁴	63 (0x3F)	D							00_000			B							0000_0									Rc
fsqrtsx ⁴	59 (0x3B)	D							00_000			B							0000_0									Rc
fsubx	63 (0x3F)	D							A			B							0000_0									Rc
fsubsx	59 (0x3B)	D							A			B							0000_0									Rc
icbi	31 (0x1F)		000_00						A			B																0
isync	19 (0x13)		000_00						00_000			0_0000																0
lbz	34 (0x22)	D							A																			
lbzu	35 (0x23)	D							A																			
lbzux	31 (0x1F)	D							A			B																0
lbzx	31 (0x1F)	D							A			B																0
lfd	50 (0x32)	D							A																			
lfdv	51 (0x33)	D							A																			
lfdvx	31 (0x1F)	D							A			B																0
lfdx	31 (0x1F)	D							A			B																0
lfs	48 (0x30)	D							A																			
lfsu	49 (0x31)	D							A																			
lfsux	31 (0x1F)	D							A			B																0
lfsx	31 (0x1F)	D							A			B																0
lha	42 (0x2A)	D							A																			
lhau	43 (0x2B)	D							A																			
lhaux	31 (0x1F)	D							A			B																0
lhax	31 (0x1F)	D							A			B																0
lhbrx	31 (0x1F)	D							A			B																0
lhz	40 (0x28)	D							A																			
lhzu	41 (0x29)	D							A																			
lhzux	31 (0x1F)	D							A			B																0
lhzx	31 (0x1F)	D							A			B																0
lmw ⁵	46 (0x2E)	D							A																			
lswi ⁵	31 (0x1F)	D							A			NB																0
lswx ⁵	31 (0x1F)	D							A			B																0
lvebx ³	31 (0x1F)	vD							A			B																0
lvehx ³	31 (0x1F)	vD							A			B																0
lvewx ³	31 (0x1F)	vD							A			B																0

Table A-1. Instructions by Mnemonic (Dec, Hex) (continued)

Name	0	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
lvs^l	31 (0x1F)	vD		A		B		0006 (0x006)						0														
lvs^r	31 (0x1F)	vD		A		B		0038 (0x026)						0														
lvx^l	31 (0x1F)	vD		A		B		0103 (0x067)						0														
lvx^l	31 (0x1F)	vD		A		B		0359 (0x167)						0														
lwarx	31 (0x1F)	D		A		B		0020 (0x014)						0														
lwbrx	31 (0x1F)	D		A		B		0534 (0x216)						0														
lwz	32 (0x20)	D		A		d																						
lwzu	33 (0x21)	D		A		d																						
lwzux	31 (0x1F)	D		A		B		0055 (0x037)						0														
lwzx	31 (0x1F)	D		A		B		0023 (0x017)						0														
mcrf	19 (0x13)	crfD	00	crfS	00	0_0000						0000 (0x000)	0															
mcrfs	63 (0x3F)	crfD	00	crfS	00	0_0000						0064 (0x040)	0															
mcrxr	31 (0x1F)	crfD	00	00_000		0_0000				0512 (0x200)						0												
mfc^r	31 (0x1F)	D		00_000		0_0000				0019 (0x013)						0												
mffs^x	63 (0x3F)	D		00_000		0_0000				0583 (0x247)						Rc												
mfms^r	31 (0x1F)	D		00_000		0_0000				0083 (0x053)						0												
mfsp^r	31 (0x1F)	D		spr										0339 (0x153)	0													
mfs^r	31 (0x1F)	D		0	SR		0_0000				0595 (0x099)						0											
mfsrin²	31 (0x1F)	D		00_000		B				0659 (0x293)						0												
mftb	31 (0x1F)	D		tbr										0371 (0x173)	0													
mfvscr³	04 (0x04)	vD		00_000		0_0000				1540 (0x604)						0												
mtrcf	31 (0x1F)	S		0	CRM						0	0144 (0x090)						0										
mtfsb0^x	63 (0x3F)	crbD		00_000		0_0000				0070 (0x046)						Rc												
mtfsb1^x	63 (0x3F)	crbD		00_000		0_0000				0038 (0x026)						Rc												
mtfs^x	63 (0x3F)	0	FM						0	B		0711 (0x2C7)						Rc										
mtfsfi^x	63 (0x3F)	crfD	00	00_000		IMM		0	0134 (0x086)						Rc													
mtms^r	31 (0x1F)	S		00_000		0_0000				0146 (0x092)						0												
mtspr⁶	31 (0x1F)	S		spr										0467 (0x1D3)	0													
mts^r	31 (0x1F)	S		0	SR		0_0000				0210 (0x001)						0											
mtsrin²	31 (0x1F)	S		00_000		B				0242 (0x0F2)						0												
mtvscr³	04 (0x04)	000_00		00_000		vB		1604 (0x644)						0														
mulhw^x	31 (0x1F)	D		A		B		0	0075 (0x04B)						Rc													
mulhwu^x	31 (0x1F)	D		A		B		0	0011 (0x00B)						Rc													

Table A-1. Instructions by Mnemonic (Dec, Hex) (continued)

Name	0	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31				
mulli	07 (0x07)				D					A											SIMM											
mullwx	31 (0x1F)				D					A			B		OE						0235 (0x0EB)					Rc						
nandx	31 (0x1F)				S					A			B								0476 (0x1DC)					Rc						
negx	31 (0x1F)				D					A		0_0000		OE							0104 (0x068)					Rc						
norx	31 (0x1F)				S					A			B								0124 (0x07C)					Rc						
orx	31 (0x1F)				S					A			B								0444 (0x1BC)					Rc						
orcx	31 (0x1F)				S					A			B								0412 (0x19C)					Rc						
ori	24 (0x18)				S					A											UIMM											
oris	25 (0x19)				S					A											UIMM											
rfi²	19 (0x13)					000_00			00_000				0_0000								0050 (0x032)					0						
rlwimix	20 (0x14)				S					A			SH								MB			ME			Rc					
rlwinmx	21 (0x15)				S					A			SH								MB			ME			Rc					
rlwnmx	23 (0x17)				S					A			B								MB			ME			Rc					
sc	17 (0x11)					000_0000_0000_0000_0000_0000_00																								1		0
slwx	31 (0x1F)				S					A			B								0024 (0x018)					Rc						
srawx	31 (0x1F)				S					A			B								0792 (0x318)					Rc						
srawix	31 (0x1F)				S					A			SH								0824 (0x338)					Rc						
srwx	31 (0x1F)				S					A			B								0536 (0x218)					Rc						
stb	38 (0x26)				S					A											d											
stbu	39 (0x27)				S					A											d											
stbux	31 (0x1F)				S					A			B								0247 (0x0F7)					0						
stbx	31 (0x1F)				S					A			B								0215 (0x0D7)					0						
stfd	54 (0x36)				S					A											d											
stfdu	55 (0x37)				S					A											d											
stfdux	31 (0x1F)				S					A			B								0759 (0x2F7)					0						
stfdx	31 (0x1F)				S					A			B								0727 (0x2D7)					0						
stfiwx¹	31 (0x1F)				S					A			B								0983 (0x3D7)					0						
stfs	52 (0x34)				S					A											d											
stfsu	53 (0x35)				S					A											d											
stfsux	31 (0x1F)				S					A			B								0695 (0x2B7)					0						
stfsx	31 (0x1F)				S					A			B								0663 (0x297)					0						
sth	44 (0x2C)				S					A											d											
sthbrx	31 (0x1F)				S					A			B								0918 (0x396)					0						

Table A-1. Instructions by Mnemonic (Dec, Hex) (continued)

Name	0	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	
vctuxs ³	04 (0x04)		vD						UIMM										vB										0906 (0x38A)
vexptefp ³	04 (0x04)		vD						00_000										vB										0394 (0x18A)
vlogefz ³	04 (0x04)		vD						00_000										vB										0458 (0x1CA)
vmaddfp ³	04 (0x04)		vD						vA										vB		vC								0046 (0x002E)
vmaxfp ³	04 (0x04)		vD						vA										vB										1034 (0x040A)
vmaxsb ³	04 (0x04)		vD						vA										vB										0258 (0x028)
vmaxsh ³	04 (0x04)		vD						vA										vB										0322 (0x01C)
vmaxsw ³	04 (0x04)		vD						vA										vB										0386 (0x182)
vmaxub ³	04 (0x04)		vD						vA										vB										0002 (0x002)
vmaxuh ³	04 (0x04)		vD						vA										vB										0066 (0x042)
vmaxuw ³	04 (0x04)		vD						vA										vB										0130 (0x082)
vmhaddshs ³	04 (0x04)		vD						vA										vB		vC								0032 (0x020)
vmhraddshs ³	04 (0x04)		vD						vA										vB		vC								0033 (0x021)
vminfp ³	04 (0x04)		vD						vA										vB										1098 (0x44A)
vminsb ³	04 (0x04)		vD						vA										vB										0770 (0x302)
vminsh ³	04 (0x04)		vD						vA										vB										0834 (0x342)
vminsw ³	04 (0x04)		vD						vA										vB										0898 (0x382)
vminub ³	04 (0x04)		vD						vA										vB										0514 (0x202)
vminuh ³	04 (0x04)		vD						vA										vB										0578 (0x242)
vminuw ³	04 (0x04)		vD						vA										vB										0642 (0x282)
vmladduhm ³	04 (0x04)		vD						vA										vB		vC								0034 (0x022)
vmrghb ³	04 (0x04)		vD						vA										vB										0012 (0x00C)
vmrghh ³	04 (0x04)		vD						vA										vB										0076 (0x04C)
vmrghw ³	04 (0x04)		vD						vA										vB										0140 (0x08C)
vmrglb ³	04 (0x04)		vD						vA										vB										0268 (0x008)
vmrglh ³	04 (0x04)		vD						vA										vB										0332 (0x14C)
vmrglw ³	04 (0x04)		vD						vA										vB										0396 (0x18C)
vmsummbm ³	04 (0x04)		vD						vA										vB		vC								0037 (0x025)
vmsumshm ³	04 (0x04)		vD						vA										vB		vC								0040 (0x028)
vmsumshs ³	04 (0x04)		vD						vA										vB		vC								0041 (0x029)
vmsumubm ³	04 (0x04)		vD						vA										vB		vC								0036 (0x024)
vmsumuhm ³	04 (0x04)		vD						vA										vB		vC								0038 (0x026)
vmsumuhs ³	04 (0x04)		vD						vA										vB		vC								0039 (0x027)

Table A-1. Instructions by Mnemonic (Dec, Hex) (continued)

Name	0	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
vsldoi ³	04 (0x04)	vD	vA	vB	0	SH	0044 (0x02C)																					
vslh ³	04 (0x04)	vD	vA	vB	0324 (0x144)																							
vslo ³	04 (0x04)	vD	vA	vB	1036 (0x40C)																							
vslw ³	04 (0x04)	vD	vA	vB	0388 (0x184)																							
vspltb ³	04 (0x04)	vD	UIMM	vB	0524 (0x20C)																							
vsplth ³	04 (0x04)	vD	UIMM	vB	0588 (0x24C)																							
vspltisb ³	04 (0x04)	vD	SIMM	0_0000	0780 (0x30C)																							
vspltish ³	04 (0x04)	vD	SIMM	0_0000	0844 (0x34C)																							
vspltisw ³	04 (0x04)	vD	SIMM	0_0000	0908 (0x38C)																							
vspltw ³	04 (0x04)	vD	UIMM	vB	0652 (0x28C)																							
vsr ³	04 (0x04)	vD	vA	vB	0708 (0x2C4)																							
vsrab ³	04 (0x04)	vD	vA	vB	0772 (0x304)																							
vsrah ³	04 (0x04)	vD	vA	vB	0836 (0x344)																							
vsraw ³	04 (0x04)	vD	vA	vB	0900 (0x384)																							
vsrb ³	04 (0x04)	vD	vA	vB	0516 (0x204)																							
vsrh ³	04 (0x04)	vD	vA	vB	0580 (0x244)																							
vsro ³	04 (0x04)	vD	vA	vB	1100 (0x44C)																							
vsrw ³	04 (0x04)	vD	vA	vB	0644 (0x284)																							
vsubcuw ³	04 (0x04)	vD	vA	vB	1408 (0x580)																							
vsubfp ³	04 (0x04)	vD	vA	vB	0074 (0x4A)																							
vsubsb ³	04 (0x04)	vD	vA	vB	1792 (0x700)																							
vsubsh ³	04 (0x04)	vD	vA	vB	1856 (0x740)																							
vsubsw ³	04 (0x04)	vD	vA	vB	1920 (0x780)																							
vsubbm ³	04 (0x04)	vD	vA	vB	1024 (0x400)																							
vsubub ³	04 (0x04)	vD	vA	vB	1536 (0x600)																							
vsubuh ³	04 (0x04)	vD	vA	vB	1088 (0x440)																							
vsubuhs ³	04 (0x04)	vD	vA	vB	1600 (0x640)																							
vsubuwm ³	04 (0x04)	vD	vA	vB	1152 (0x480)																							
vsubuws ³	04 (0x04)	vD	vA	vB	1664 (0x680)																							
vsumsw ³	04 (0x04)	vD	vA	vB	1928 (0x788)																							
vsum2sw ³	04 (0x04)	vD	vA	vB	1672 (0x688)																							
vsum4sb ³	04 (0x04)	vD	vA	vB	1800 (0x708)																							
vsum4sh ³	04 (0x04)	vD	vA	vB	1608 (0x648)																							

Table A-1. Instructions by Mnemonic (Dec, Hex) (continued)

Name	0	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31		
vsum4ubs ³	04 (0x04)		vD				vA								vB														1544 (0x608)	
vupkhp ³	04 (0x04)		vD				00_000								vB														0846 (0x34E)	
vupkhsb ³	04 (0x04)		vD				00_000								vB														0526 (0x20E)	
vupksh ³	04 (0x04)		vD				00_000								vB														0590 (0x24E)	
vupklp ³	04 (0x04)		vD				00_000								vB														0974 (0x3CE)	
vupklsb ³	04 (0x04)		vD				00_000								vB														0654 (0x28E)	
vupkls ³	04 (0x04)		vD				00_000								vB														0718 (0x2CE)	
vxor ³	04 (0x04)		vD				vA								vB														1220 (0x4C4)	
xorx	31 (0x1F)		S				A								B														0316 (0x13C)	Rc
xori	26 (0x1A)		S				A																						UIMM	
xoris	27 (0x1B)		S				A																						UIMM	

- ¹ Optional to the PowerPC architecture but implemented by the e600 core.
- ² Supervisor-level instruction.
- ³ AltiVec technology-specific instruction.
- ⁴ Optional instruction not implemented by the e600 core.
- ⁵ Load/store string/multiple instruction.
- ⁶ Supervisor- and user-level instruction.

A.2 Instructions Sorted by Primary and Secondary Opcodes (Decimal and Hexadecimal)

Table A-2 shows the instructions implemented in the e600 core. The instructions are listed by their primary (0–5) and secondary (21–31) opcodes in decimal and hexadecimal format.

Key:
 Reserved bits

Table A-2. Instructions by Primary and Secondary Opcodes (Dec, Hex)

Name	0	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31			
twi	03 (0x03)				TO																								SIMM		
vaddubm ¹	04 (0x04)		vD				vA								vB														0000 (0x000)	0	
vmaxub ¹	04 (0x04)		vD				vA								vB															0002 (0x002)	
vrlb ¹	04 (0x04)		vD				vA								vB															0004 (0x004)	
vcmpequb ¹	04 (0x04)		vD				vA								vB															0006 (0x006)	Rc
vmuloub ¹	04 (0x04)		vD				vA								vB															0008 (0x008)	
vaddfp ¹	04 (0x04)		vD				vA								vB															0010 (0x0B4)	0

Table A-2. Instructions by Primary and Secondary Opcodes (Dec, Hex) (continued)

Name	0	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	
vmrghb ¹	04 (0x04)	vD								vA											vB								0012 (0x00C)
vpkuhum ¹	04 (0x04)	vD								vA											vB								0014 (0x00E)
vmhaddshs ¹	04 (0x04)	vD								vA											vB	vC							0032 (0x020)
vmhraddshs ¹	04 (0x04)	vD								vA											vB	vC							0033 (0x021)
vmladduhm ¹	04 (0x04)	vD								vA											vB	vC							0034 (0x022)
vmsumubm ¹	04 (0x04)	vD								vA											vB	vC							0036 (0x024)
vmsummbm ¹	04 (0x04)	vD								vA											vB	vC							0037 (0x025)
vmsumuhm ¹	04 (0x04)	vD								vA											vB	vC							0038 (0x026)
vmsumuhs ¹	04 (0x04)	vD								vA											vB	vC							0039 (0x027)
vmsumshm ¹	04 (0x04)	vD								vA											vB	vC							0040 (0x028)
vmsumshs ¹	04 (0x04)	vD								vA											vB	vC							0041 (0x029)
vsel ¹	04 (0x04)	vD								vA											vB	vC							0042 (0x02A)
vperm ¹	04 (0x04)	vD								vA											vB	vC							0043 (0x02B)
vsldoi ¹	04 (0x04)	vD								vA											vB	0	SH						0044 (0x02C)
vmaddfp ¹	04 (0x04)	vD								vA											vB	vC							0046 (0x002E)
vnmsubfp ¹	04 (0x04)	vD								vA											vB	vC							0047 (0x02F)
vadduhm ¹	04 (0x04)	vD								vA											vB								0064 (0x040) 0
vmaxuh ¹	04 (0x04)	vD								vA											vB								0066 (0x042)
vrlh ¹	04 (0x04)	vD								vA											vB								0068 (0x044)
vcmpquhx ¹	04 (0x04)	vD								vA											vB	Rc							0070 (0x046)
vmulouh ¹	04 (0x04)	vD								vA											vB								0072 (0x048)
vsubfp ¹	04 (0x04)	vD								vA											vB								0074 (0x4A)
vmrghh ¹	04 (0x04)	vD								vA											vB								0076 (0x04C)
vpkuwum ¹	04 (0x04)	vD								vA											vB								0078 (0x04E)
vadduw ¹	04 (0x04)	vD								vA											vB								0128 (0x0F0) 0
vmaxuw ¹	04 (0x04)	vD								vA											vB								0130 (0x082)
vrlw ¹	04 (0x04)	vD								vA											vB								0132 (0x084)
vcmpquw ¹	04 (0x04)	vD								vA											vB	Rc							0134 (0x086)
vmrghw ¹	04 (0x04)	vD								vA											vB								0140 (0x08C)
vpkuhus ¹	04 (0x04)	vD								vA											vB								0142 (0x08E)
vcmpqfp ¹	04 (0x04)	vD								vA											vB	Rc							0198 (0x0C6)
vpkuwus ¹	04 (0x04)	vD								vA											vB								0206 (0x0CE)
vmaxsb ¹	04 (0x04)	vD								vA											vB								0258 (0x028)

Table A-2. Instructions by Primary and Secondary Opcodes (Dec, Hex) (continued)

Name	0	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31			
vslb ¹	04 (0x04)	vD							vA						vB															0260 (0x104)	
vmulosb ¹	04 (0x04)	vD							vA						vB															0264 (0x108)	
vrefp ¹	04 (0x04)	vD							00_000						vB															0266 (0x10A)	
vmrglb ¹	04 (0x04)	vD							vA						vB															0268 (0x008)	
vpkshus ¹	04 (0x04)	vD							vA						vB															0270 (0x012)	
vmaxsh ¹	04 (0x04)	vD							vA						vB															0322 (0x01C)	
vslh ¹	04 (0x04)	vD							vA						vB															0324 (0x144)	
vmulosh ¹	04 (0x04)	vD							vA						vB															0328 (0x148)	
vrsqrtefp ¹	04 (0x04)	vD							00_000						vB															0330 (0x14A)	
vmrglh ¹	04 (0x04)	vD							vA						vB															0332 (0x14C)	
vpkswus ¹	04 (0x04)	vD							vA						vB															0334 (0x14E)	
vaddcuw ¹	04 (0x04)	vD							vA						vB															0384 (0x180)	0
vmaxsw ¹	04 (0x04)	vD							vA						vB															0386 (0x182)	
vslw ¹	04 (0x04)	vD							vA						vB															0388 (0x184)	
vexpteftp ¹	04 (0x04)	vD							00_000						vB															0394 (0x18A)	
vmrglw ¹	04 (0x04)	vD							vA						vB															0396 (0x18C)	
vpkshss ¹	04 (0x04)	vD							vA						vB															0398 (0x18E)	
vsl ¹	04 (0x04)	vD							vA						vB															0452 (0x1C4)	
vcmpgefp ^{x1}	04 (0x04)	vD							vA						vB	Rc														0454 (0x1C6)	
vlogefp ¹	04 (0x04)	vD							00_000						vB															0458 (0x1CA)	
vpkswss ¹	04 (0x04)	vD							vA						vB															0462 (0x00C)	
vaddubs ¹	04 (0x04)	vD							vA						vB															0512 (0x200)	0
vminub ¹	04 (0x04)	vD							vA						vB															0514 (0x202)	
vsrb ¹	04 (0x04)	vD							vA						vB															0516 (0x204)	
vcmpgtub ^{x1}	04 (0x04)	vD							vA						vB	Rc														0518 (0x206)	
vmuleub ¹	04 (0x04)	vD							vA						vB															0520 (0x208)	
vrfin ¹	04 (0x04)	vD							00_000						vB															0522 (0x20A)	
vspltb ¹	04 (0x04)	vD							UIMM						vB															0524 (0x20C)	
vupkhsb ¹	04 (0x04)	vD							00_000						vB															0526 (0x20E)	
vadduhs ¹	04 (0x04)	vD							vA						vB															0576 (0x240)	0
vminuh ¹	04 (0x04)	vD							vA						vB															0578 (0x242)	
vsrh ¹	04 (0x04)	vD							vA						vB															0580 (0x244)	
vcmpgtuh ^{x1}	04 (0x04)	vD							vA						vB	Rc														0582 (0x246)	

Table A-2. Instructions by Primary and Secondary Opcodes (Dec, Hex) (continued)

Name	0	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	
vsraw ¹	04 (0x04)	vD	vA	vB	0900 (0x384)																								
vcmpgtsw ¹	04 (0x04)	vD	vA	vB	Rc	0902 (0x386)																							
vctuxs ¹	04 (0x04)	vD	UIMM	vB	0906 (0x38A)																								
vspltisw ¹	04 (0x04)	vD	SIMM	0_0000	0908 (0x38C)																								
vcmpbfp ¹	04 (0x04)	vD	vA	vB	Rc	0966 (0x3C6)																							
vctxs ¹	04 (0x04)	vD	UIMM	vB	0970 (0x3CA)																								
vupklpx ¹	04 (0x04)	vD	00_000	vB	0974 (0x3CE)																								
vsububm ¹	04 (0x04)	vD	vA	vB	1024 (0x400)																								
vavgub ¹	04 (0x04)	vD	vA	vB	1026 (0x402)																								0
vand ¹	04 (0x04)	vD	vA	vB	1028 (0x118)																								0
vmaxfp ¹	04 (0x04)	vD	vA	vB	1034 (0x040A)																								
vslo ¹	04 (0x04)	vD	vA	vB	1036 (0x40C)																								
vsubuhm ¹	04 (0x04)	vD	vA	vB	1088 (0x440)																								
vavguh ¹	04 (0x04)	vD	vA	vB	1090 (0x442)																								0
vandc ¹	04 (0x04)	vD	vA	vB	1092 (0x444)																								0
vminf ¹	04 (0x04)	vD	vA	vB	1098 (0x44A)																								
vsro ¹	04 (0x04)	vD	vA	vB	1100 (0x44C)																								
vsubuwm ¹	04 (0x04)	vD	vA	vB	1152 (0x480)																								
vavguw ¹	04 (0x04)	vD	vA	vB	1154 (0x482)																								0
vor ¹	04 (0x04)	vD	vA	vB	1156 (0x484)																								
vxor ¹	04 (0x04)	vD	vA	vB	1220 (0x4C4)																								
vavgsb ¹	04 (0x04)	vD	vA	vB	1282 (0x502)																								0
vnor ¹	04 (0x04)	vD	vA	vB	1284 (0x504)																								
vavgsh ¹	04 (0x04)	vD	vA	vB	1346 (0x542)																								0
vsubcuw ¹	04 (0x04)	vD	vA	vB	1408 (0x580)																								
vavgsw ¹	04 (0x04)	vD	vA	vB	1410 (0x582)																								0
vsububs ¹	04 (0x04)	vD	vA	vB	1536 (0x600)																								
mfvscr ¹	04 (0x04)	vD	00_000	0_0000	1540 (0x604)																								0
vsum4ubs ¹	04 (0x04)	vD	vA	vB	1544 (0x608)																								
vsubuhs ¹	04 (0x04)	vD	vA	vB	1600 (0x640)																								
mtvscr ¹	04 (0x04)	000_00	00_000	vB	1604 (0x644)																								0
vsum4shs ¹	04 (0x04)	vD	vA	vB	1608 (0x648)																								
vsubuws ¹	04 (0x04)	vD	vA	vB	1664 (0x680)																								

Table A-2. Instructions by Primary and Secondary Opcodes (Dec, Hex) (continued)

Name	0	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
vsum2sws ¹	04 (0x04)	vD		vA		vB		1672 (0x688)																				
vsubsbs ¹	04 (0x04)	vD		vA		vB		1792 (0x700)																				
vsum4sbs ¹	04 (0x04)	vD		vA		vB		1800 (0x708)																				
vsubshs ¹	04 (0x04)	vD		vA		vB		1856 (0x740)																				
vsubsws ¹	04 (0x04)	vD		vA		vB		1920 (0x780)																				
vsumsws ¹	04 (0x04)	vD		vA		vB		1928 (0x788)																				
mulli	07 (0x07)	D		A		SIMM																						
subfic	08 (0x08)	D		A		SIMM																						
cmpli	10 (0x0A)	crfD	0	L	A		UIMM																					
cmpi	11 (0x0B)	crfD	0	L	A		SIMM																					
addic	12 (0xC)	D		A		SIMM																						
addic.	13 (0xD)	D		A		SIMM																						
addi	14 (0xE)	D		A		SIMM																						
addis	15 (0xF)	D		A		SIMM																						
bcx	16 (0x10)	BO		BI		BD																			AA	LK		
sc	17 (0x11)	000_0000_0000_0000_0000_00																								1	0	
bx	18 (0x12)	LI																								AA	LK	
mcrf	19 (0x13)	crfD	00	crfS	00	0_0000		0000 (0x000)														0						
bclrx	19 (0x13)	BO		BI		0_0000		0016 (0x010)														LK						
crnor	19 (0x13)	crbD		crbA		crbB		0033 (0x21)														0						
rfi ²	19 (0x13)	000_00		00_000		0_0000		0050 (0x032)														0						
crandc	19 (0x13)	crbD		crbA		crbB		0129 (0x081)														0						
isync	19 (0x13)	000_00		00_000		0_0000		0150 (0x096)														0						
crxor	19 (0x13)	crbD		crbA		crbB		0193 (0C1)														0						
crnand	19 (0x13)	crbD		crbA		crbB		0225 (0x0E1)														0						
crand	19 (0x13)	crbD		crbA		crbB		0257 (0x101)														0						
creqv	19 (0x13)	crbD		crbA		crbB		0289 (0x121)														0						
crorc	19 (0x13)	crbD		crbA		crbB		0417 (0x1A1)														0						
cror	19 (0x13)	crbD		crbA		crbB		0449 (0x1C1)														0						
bcctrx	19 (0x13)	BO		BI		0_0000		0528 (0x210)														LK						
rlwimix	20 (0x14)	S		A		SH		MB		ME		Rc																
rlwinmx	21 (0x15)	S		A		SH		MB		ME		Rc																
rlwnmx	23 (0x17)	S		A		B		MB		ME		Rc																

Table A-2. Instructions by Primary and Secondary Opcodes (Dec, Hex) (continued)

Name	0	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
ori	24 (0x18)	S			A			UIMM																				
oris	25 (0x19)	S			A			UIMM																				
xori	26 (0x1A)	S			A			UIMM																				
xoris	27 (0x1B)	S			A			UIMM																				
andi.	28 (0x1C)	S			A			UIMM																				
andis.	29 (0x1D)	S			A			UIMM																				
cmp	31 (0x1F)	crfD	0	L	A			B			0000 (0x000)						0											
tw	31 (0x1F)	TO			A			B			0004 (0x004)						0											
lvs¹	31 (0x1F)	vD			A			B			0006 (0x006)						0											
lvebx¹	31 (0x1F)	vD			A			B			0007 (0x007)						0											
subfcx	31 (0x1F)	D			A			B			OE	0008 (0x008)						Rc										
addcx	31 (0x1F)	D			A			B			OE	0010 (0x00A)						Rc										
mulhwux	31 (0x1F)	D			A			B			0	0011 (0x00B)						Rc										
mfc^r	31 (0x1F)	D			00_000			0_0000			0019 (0x013)						0											
lwarx	31 (0x1F)	D			A			B			0020 (0x014)						0											
lwzx	31 (0x1F)	D			A			B			0023 (0x017)						0											
slwx	31 (0x1F)	S			A			B			0024 (0x018)						Rc											
cntlzwx	31 (0x1F)	S			A			0_0000			0026 (0x01A)						Rc											
andx	31 (0x1F)	S			A			B			0028 (0x01C)						Rc											
cmpl	31 (0x1F)	crfD	0	L	A			B			0032 (0x020)						0											
lvs^{r1}	31 (0x1F)	vD			A			B			0038 (0x026)						0											
lvehx¹	31 (0x1F)	vD			A			B			0039 (0x027)						0											
subfx	31 (0x1F)	D			A			B			OE	0040 (0x028)						Rc										
dcbst	31 (0x1F)	000_00			A			B			0054 (0x036)						0											
lwzux	31 (0x1F)	D			A			B			0055 (0x037)						0											
andcx	31 (0x1F)	S			A			B			0060 (0x03C)						Rc											
lvewx¹	31 (0x1F)	vD			A			B			0071 (0x047)						0											
mulhw^x	31(0x1F)	D			A			B			0	0075 (0x04B)						Rc										
mfms^{r2}	31 (0x1F)	D			00_000			0_0000			0083 (0x053)						0											
dcbf	31 (0x1F)	000_00			A			B			0086 (0x056)						0											
lbzx	31 (0x1F)	D			A			B			0087 (0x057)						0											
lvx¹	31 (0x1F)	vD			A			B			0103 (0x067)						0											
negx	31 (0x1F)	D			A			0_0000			OE	0104 (0x068)						Rc										

Table A-2. Instructions by Primary and Secondary Opcodes (Dec, Hex) (continued)

Name	0	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	
lbzux	31 (0x1F)				D					A					B														0
norx	31 (0x1F)				S					A					B														Rc
stvebx¹	31 (0x1F)				vS					A					B														0
subfex	31 (0x1F)				D					A					B			OE											Rc
addex	31 (0x1F)				D					A					B			OE											Rc
mtrcf	31 (0x1F)				S		0						CRM				0												0
mtmsr²	31 (0x1F)				S				00_000						0_0000														0
stwcx.	31 (0x1F)				S					A					B														1
stwx	31 (0x1F)				S					A					B														0
stvehx¹	31 (0x1F)				vS					A					B														0
stwux	31 (0x1F)				S					A					B														0
stvewx¹	31 (0x1F)				vS					A					B														0
subfzex	31 (0x1F)				D					A					0_0000			OE											Rc
addzex	31 (0x1F)				D					A					0_0000			OE											Rc
mtsr²	31 (0x1F)				S		0			SR					0_0000														0
stbx	31 (0x1F)				S					A					B														0
stvx¹	31 (0x1F)				vS					A					B														0
subfmex	31 (0x1F)				D					A					0_0000			OE											Rc
addmex	31 (0x1F)				D					A					0_0000			OE											Rc
mullwx	31 (0x1F)				D					A					B			OE											Rc
mtsrin²	31 (0x1F)				S				00_000						B														0
dcbtst	31 (0x1F)					000_00				A					B														0
stbux	31 (0x1F)				S					A					B														0
addx	31 (0x1F)				D					A					B			OE											Rc
dcbt	31 (0x1F)					000_00				A					B														0
lhzx	31 (0x1F)				D					A					B														0
eqvx	31 (0x1F)				S					A					B														Rc
tlbie^{2,3}	31 (0x1F)					000_00			00_000						B														0
eciwx³	31 (0x1F)				D					A					B														0
lhzux	31 (0x1F)				D					A					B														0
xorx	31 (0x1F)				S					A					B														Rc
mfspi⁴	31 (0x1F)				D									spr															0
dst¹	31 (0x1F)	0		00	STRM					A				B															0

Table A-2. Instructions by Primary and Secondary Opcodes (Dec, Hex) (continued)

Name	0	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31		
dstt ¹	31 (0x1F)	1	00	STRM			A								B														0	
lhax	31 (0x1F)				D			A							B														0	
lvxl ¹	31 (0x1F)				vD			A							B														0	
tlbia ⁵	31 (0x1F)				000_00			00_000							0_0000														0	
mftb	31 (0x1F)				D										tbr														0	
dstst ¹	31 (0x1F)	0	00	STRM			A								B														0	
dststt ¹	31 (0x1F)	1	00	STRM			A								B														0	
lhax	31 (0x1F)				D			A							B														0	
sthx	31 (0x1F)				S			A							B														0	
orc_x	31 (0x1F)				S			A							B														Rc	
ecow_x ³	31 (0x1F)				S			A							B														0	
sthux	31 (0x1F)				S			A							B														0	
or_x	31 (0x1F)				S			A							B														Rc	
divw_x	31 (0x1F)				D			A							B		OE												Rc	
mtspr ⁴	31 (0x1F)				S										spr														0	
dcbl ²	31 (0x1F)				000_00			A							B														0	
nand_x	31 (0x1F)				S			A							B														Rc	
stvxl ¹	31 (0x1F)				vS			A							B														0	
divw_x	31 (0x1F)				D			A							B		OE												Rc	
mcrxr	31 (0x1F)		crfD		00			00_000							0_0000															0
lsw_x ⁶	31 (0x1F)				D			A							B														0	
lwbr_x	31 (0x1F)				D			A							B														0	
lfs_x	31 (0x1F)				D			A							B														0	
srw_x	31 (0x1F)				S			A							B														Rc	
tlbsync ^{2,3}	31 (0x1F)				000_00			00_000							0_0000															0
lfsux	31 (0x1F)				D			A							B														0	
mfsr ²	31 (0x1F)				D		0	SR							0_0000														0	
lswl ⁶	31 (0x1F)				D			A							NB														0	
sync	31 (0x1F)				000_00			00_000							0_0000															0
lfd_x	31 (0x1F)				D			A							B														0	
lfd_{ux}	31 (0x1F)				D			A							B														0	
mfsrin ²	31 (0x1F)				D			00_000							B														0	
stsw_x ⁶	31 (0x1F)				S			A							B														0	

Table A-2. Instructions by Primary and Secondary Opcodes (Dec, Hex) (continued)

Name	0	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	
stwbrx	31 (0x1F)		S						A						B							0662 (0x296)							0
stfsx	31 (0x1F)		S						A						B							0663 (0x297)							0
stfsux	31 (0x1F)		S						A						B							0695 (0x2B7)							0
stswi⁶	31 (0x1F)		S						A						NB							0725 (0x2D5)							0
stfdx	31 (0x1F)		S						A						B							0727 (0x2D7)							0
dcba³	31 (0x1F)				000_00				A						B							0758 (0x2F6)							0
stfdux	31 (0x1F)		S						A						B							0759 (0x2F7)							0
lhbrx	31 (0x1F)				D				A						B							0790 (0x316)							0
srawx	31 (0x1F)				S				A						B							0792 (0x318)							Rc
dss¹	31 (0x1F)	0		00	STRM				00_000						0_0000							0822 (0x336)							0
dssall¹	31 (0x1F)	1		00	STRM				00_000						0_0000							0822 (0x336)							0
srawix	31 (0x1F)				S				A						SH							0824 (0x338)							Rc
eieio	31 (0x1F)				000_00				00_000						0_0000							0854 (0x356)							0
sthbrx	31 (0x1F)		S						A						B							0918 (0x396)							0
extshx	31 (0x1F)		S						A						0_0000							0922 (0x39A)							Rc
extsbx	31 (0x1F)		S						A						0_0000							0954 (0x3BA)							Rc
tlbld^{2,3}	31 (0x1F)				000_00				00_000						B							0978 (0x3D2)							0
icbi	31 (0x1F)				000_00				A						B							0982 (0x3D6)							0
stfiwx³	31 (0x1F)		S						A						B							0983 (0x3D7)							0
tlbli^{2,3}	31 (0x1F)				000_00				00_000						B							1010 (0x3F2)							0
dcbz	31 (0x1F)				000_00				A						B							1014 (0x3F6)							0
lwz	32 (0x20)				D				A													d							
lwzu	33 (0x21)				D				A													d							
lbz	34 (0x22)				D				A													d							
lbzu	35 (0x23)				D				A													d							
stw	36 (0x24)				S				A													d							
stwu	37 (0x25)				S				A													d							
stb	38 (0x26)				S				A													d							
stbu	39 (0x27)				S				A													d							
lhz	40 (0x28)				D				A													d							
lhzu	41 (0x29)				D				A													d							
lha	42 (0x2A)				D				A													d							
lhau	43 (0x2B)				D				A													d							

Table A-2. Instructions by Primary and Secondary Opcodes (Dec, Hex) (continued)

Name	0	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	
sth	44 (0x2C)				S					A																			d
sthu	45 (0x2D)				S					A																			d
lmw⁶	46 (0x2E)				D					A																			d
stmw⁶	47 (0x2F)				S					A																			d
lfs	48 (0x30)				D					A																			d
lfsu	49 (0x31)				D					A																			d
lfd	50 (0x32)				D					A																			d
lfdv	51 (0x33)				D					A																			d
stfs	52 (0x34)				S					A																			d
stfsu	53 (0x35)				S					A																			d
stfd	54 (0x36)				S					A																			d
stfdv	55 (0x37)				S					A																			d
fdv_x	59 (0x3B)				D					A				B				0000_0				0018 (0x012)						Rc	
fsub_x	59 (0x3B)				D					A				B				0000_0				0020 (0x014)							Rc
fadd_x	59 (0x3B)				D					A				B				0000_0				0021 (0x015)							Rc
fsqrt_x⁵	59 (0x3B)				D					00_000				B				0000_0				0022 (0x016)							Rc
fres_x³	59 (0x3B)				D					00_000				B				0000_0				0024 (0x018)							Rc
fmul_x	59 (0x3B)				D					A				0_0000				C				0025 (0x019)							Rc
fmsub_x	59 (0x3B)				D					A				B				C				0028 (0x01C)							Rc
fmadd_x	59 (0x3B)				D					A				B				C				0029 (0x01D)							Rc
fnmsub_x	59 (0x3B)				D					A				B				C				0030 (0x01E)							Rc
fnmadd_x	59 (0x3B)				D					A				B				C				0031 (0x01F)							Rc
fcmpu	63 (0x3F)				crfD		00			A				B				0000 (0x000)										0	
frsp_x	63 (0x3F)				D					00_000				B				0012 (0xC)											Rc
fctiw_x	63 (0x3F)				D					00_000				B				0014 (0x00E)											Rc
fctiwz_x	63 (0x3F)				D					00_000				B				0015 (0x00F)											Rc
fdv_x	63 (0x3F)				D					A				B				0000_0				0018 (0x012)							Rc
fsub_x	63 (0x3F)				D					A				B				0000_0				0020 (0x014)							Rc
fadd_x	63 (0x3F)				D					A				B				0000_0				0021 (0x015)							Rc
fsqrt_x⁵	63 (0x3F)				D					00_000				B				0000_0				0022 (0x016)							Rc
fsel_x³	63 (0x3F)				D					A				B				C				0023 (0x017)							Rc
fmul_x	63 (0x3F)				D					A				0_0000				C				0025 (0x019)							Rc
frsqrt_x³	63 (0x3F)				D					00_000				B				0000_0				0026 (0x01A)							Rc

Table A-2. Instructions by Primary and Secondary Opcodes (Dec, Hex) (continued)

Name	0	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
fmsub_x	63 (0x3F)	D			A			B			C			0028 (0x01C)			Rc											
fmadd_x	63 (0x3F)	D			A			B			C			0029 (0x01D)			Rc											
fnmsub_x	63 (0x3F)	D			A			B			C			0030 (0x01E)			Rc											
fnmadd_x	63 (0x3F)	D			A			B			C			0031 (0x01F)			Rc											
fcmpo	63 (0x3F)	crfD	00		A			B			0032 (0x020)			0														
mtfsb1_x	63 (0x3F)	crbD			00_000			0_0000			0038 (0x026)			Rc														
fneg_x	63 (0x3F)	D			00_000			B			0040 (0x28)			Rc														
mcrfs	63 (0x3F)	crfD	00		crfS	00		0_0000			0064 (0x040)			0														
mtfsb0_x	63 (0x3F)	crbD			00_000			0_0000			0070 (0x046)			Rc														
fmr_x	63 (0x3F)	D			00_000			B			0072 (0x48)			Rc														
mtfsfi_x	63 (0x3F)	crfD	00		00_000			IMM		0		0134 (0x086)			Rc													
fnabs_x	63 (0x3F)	D			00_000			B			0136 (0x88)			Rc														
fabs_x	63 (0x3F)	D			00_000			B			0264 (0x108)			Rc														
mffs_x	63 (0x3F)	D			00_000			0_0000			0583 (0x247)			Rc														
mtfsf_x	63 (0x3F)	0	FM				0		B			0711 (0x2C7)			Rc													

¹ AltiVec technology-specific instruction.

² Supervisor-level instruction.

³ Optional to the PowerPC architecture but implemented by the e600 core.

⁴ Supervisor- and user-level instructions.

⁵ Optional instruction not implemented by the e600 core.

⁶ Load/store string/multiple instruction.

A.3 Instructions Sorted by Mnemonic (Binary)

Table A-3 shows instructions listed in alphabetical order by mnemonic with binary values.

Key:

 Reserved bits

Table A-3. Instructions by Mnemonic (Bin)

Name	0	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
add_x	011111	D			A			B			OE	100001 010			Rc													
addc_x	011111	D			A			B			OE	000001010			Rc													
addex_x	011111	D			A			B			OE	010001010			Rc													
addi	001110	D			A			SIMM																				
addic	001100	D			A			SIMM																				

Table A-3. Instructions by Mnemonic (Bin) (continued)

Name	0	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
addic.	001101	D			A			SIMM																				
addis	001111	D			A			SIMM																				
addmex	011111	D			A			0_0000	OE	11101010						Rc												
addzex	011111	D			A			0_0000	OE	11001010						Rc												
andx	011111	S			A			B			000011100						Rc											
andcx	011111	S			A			B			000111100						Rc											
andi.	011100	S			A			UIMM																				
andis.	011101	S			A			UIMM																				
bx	010010	LI										AA	LK															
bcx	010000	BO			BI			BD						AA	LK													
bcctrx	010011	BO			BI			0_0000	1000010000						LK													
bclrx	010011	BO			BI			0_0000	0000010000						LK													
cmp	011111	crfD	0	L	A			B			0000000000						0											
cmpi	001011	crfD	0	L	A			SIMM																				
cmpl	011111	crfD	0	L	A			B			0000100000						0											
cmpli	001010	crfD	0	L	A			UIMM																				
cntlzwx	011111	S			A			0_0000	0000011010						Rc													
crand	010011	crbD			crbA			crbB			0100000001						0											
crandc	010011	crbD			crbA			crbB			0010000001						0											
creqv	010011	crbD			crbA			crbB			0100100001						0											
crnand	010011	crbD			crbA			crbB			0011100001						0											
crnor	010011	crbD			crbA			crbB			0000100001						0											
cror	010011	crbD			crbA			crbB			0111000001						0											
crorc	010011	crbD			crbA			crbB			0110100001						0											
crxor	010011	crbD			crbA			crbB			0011000001						0											
dcba¹	011111	000_00			A			B			1011110110						0											
dcbf	011111	000_00			A			B			0001010110						0											
dcbi²	011111	000_00			A			B			0111010110						0											
dcbst	011111	000_00			A			B			0000110110						0											
dcbt	011111	000_00			A			B			0100010110						0											
dcbtst	011111	000_00			A			B			0011110110						0											
dcbz	011111	000_00			A			B			1111110110						0											
divwx	011111	D			A			B			OE	11110 1011						Rc										

Table A-3. Instructions by Mnemonic (Bin) (continued)

Name	0	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
divwx	011111	D			A			B			OE	1100 1011						Rc										
dss ³	011111	A	00	STRM	00_000			0_0000			1100110110						0											
dssall ³	011111	A	00	STRM	00_000			0_0000			1100110110						0											
dst ³	011111	T	00	STRM	A			B			0101010110						0											
dstst ³	011111	T	00	STRM	A			B			0101110110						0											
dststt ³	011111	1	00	STRM	A			B			0101110110						0											
dstt ³	011111	1	00	STRM	A			B			0101010110						0											
eciwx ¹	011111	D			A			B			0100110110						0											
ecowx ¹	011111	S			A			B			0110110110						0											
eieio	011111	000_00			00_000			0_0000			1101010110						0											
eqvx	011111	S			A			B			0100011100						Rc											
extsbx	011111	S			A			0_0000			1110111010						Rc											
extshx	011111	S			A			0_0000			1110011010						Rc											
fabsx	111111	D			00_000			B			0100001000						Rc											
faddx	111111	D			A			B			0000_0		1 0101				Rc											
faddsx	111011	D			A			B			0000_0		1 0101				Rc											
fcmpo	111111	crfD	00	A			B			0000100000						0												
fcmpu	111111	crfD	00	A			B			0000000000						0												
fctiw ^x	111111	D			00_000			B			0000001110						Rc											
fctiwz ^x	111111	D			00_000			B			0000001111						Rc											
fdiv ^x	111111	D			A			B			0000_0		1 0010				Rc											
fdiv ^s ^x	111011	D			A			B			0000_0		1 0010				Rc											
fmadd ^x	111111	D			A			B			C		1 1101				Rc											
fmadd ^s ^x	111011	D			A			B			C		1 1101				Rc											
fmr ^x	111111	D			00_000			B			0001001000						Rc											
fmsub ^x	111111	D			A			B			C		1 1100				Rc											
fmsub ^s ^x	111011	D			A			B			C		1 1100				Rc											
fmul ^x	111111	D			A			0_0000			C		1 1001				Rc											
fmul ^s ^x	111011	D			A			0_0000			C		1 1001				Rc											
fnabs ^x	111111	D			00_000			B			0010001000						Rc											
fneg ^x	111111	D			00_000			B			0000101000						Rc											
fnmadd ^x	111111	D			A			B			C		1 1111				Rc											
fnmadd ^s ^x	111011	D			A			B			C		1 1111				Rc											

Table A-3. Instructions by Mnemonic (Bin) (continued)

Name	0	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31			
fnmsubx	111111	D					A			B					C						1	1	1	1				Rc			
fnmsubsx	111011	D					A			B					C						1	1	1	1				Rc			
fresx ¹	111011	D				00_000				B					0000_0						1	1	0	0				Rc			
frsp ^x	111111	D				00_000				B					0000001100													Rc			
frsqr ^{te} x ¹	111111	D				00_000				B					0000_0						1	1	0	1	0			Rc			
fsel ^x ¹	111111	D					A			B					C						1	0	1	1	1			Rc			
fsqr ^t x ⁴	111111	D				00_000				B					0000_0						1	0	1	1	0			Rc			
fsqr ^t s ^x ⁴	111011	D				00_000				B					0000_0						1	0	1	1	0			Rc			
fsub ^x	111111	D					A			B					0000_0						1	0	1	0	0			Rc			
fsub ^s x	111011	D					A			B					0000_0						1	0	1	0	0			Rc			
icbi	011111		000_00				A			B					1111010110													0			
isync	010011		000_00			00_000			0_0000						0010010110													0			
lbz	100010	D					A			d																					
lbzu	100011	D					A			d																					
lbzux	011111	D					A			B					0001110111													0			
lbzx	011111	D					A			B					0001010111													0			
lfd	110010	D					A			d																					
lfd ^u	110011	D					A			d																					
lfd ^u x	011111	D					A			B					1001110111													0			
lfd ^x	011111	D					A			B					1001010111													0			
lfs	110000	D					A			d																					
lfs ^u	110001	D					A			d																					
lfs ^u x	011111	D					A			B					1000110111													0			
lfs ^x	011111	D					A			B					1000010111													0			
lha	101010	D					A			d																					
lhau	101011	D					A			d																					
lhau ^x	011111	D					A			B					0101110111													0			
lhax	011111	D					A			B					0101010111													0			
lhbr ^x	011111	D					A			B					1100010110													0			
lhz	101000	D					A			d																					
lhzu	101001	D					A			d																					
lhzu ^x	011111	D					A			B					0100110111													0			
lhz ^x	011111	D					A			B					0100010111													0			

Table A-3. Instructions by Mnemonic (Bin) (continued)

Name	0	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
lmw ⁵	101110	D			A			d																				
lswi ⁵	011111	D			A			NB			1001010101						0											
lswx ⁵	011111	D			A			B			1000010101						0											
lvebx ³	011111	vD			A			B			0000000111						0											
lvehx ³	011111	vD			A			B			0000100111						0											
lvewx ³	011111	vD			A			B			0001000111						0											
lvs ³	011111	vD			A			B			0000000110						0											
lvsr ³	011111	vD			A			B			0000100110						0											
lvx ³	011111	vD			A			B			0001100111						0											
lvxi ³	011111	vD			A			B			0101100111						0											
lwarx	011111	D			A			B			0000010100						0											
lwbrx	011111	D			A			B			1000010110						0											
lwz	100000	D			A			d																				
lwzu	100001	D			A			d																				
lwzux	011111	D			A			B			0000110111						0											
lwzx	011111	D			A			B			0000010111						0											
mcrf	010011	crfD	00	crfS	00	0_0000			00000000						0													
mcrfs	111111	crfD	00	crfS	00	0_0000			00100000						0													
mcrxr	011111	crfD	00	00_000			0_0000			1000000000						0												
mfcrr	011111	D			00_000			0_0000			0000010011						0											
mffsx	111111	D			00_000			0_0000			1001000111						Rc											
mfmsr ²	011111	D			00_000			0_0000			0001010011						0											
mfspir ⁶	011111	D			spr										0101010011			0										
mfsr ²	011111	D			0	SR			0_0000			1001010011						0										
mfsrin ²	011111	D			00_000			B			1010010011						0											
mftb	011111	D			tbr										0101110011			0										
mfvscr ³	000100	vD			00_000			0_0000			11000000100						0											
mtcrf	011111	S			0	CRM						0	0010010000						0									
mtfsb0x	111111	crbD			00_000			0_0000			0001000110						Rc											
mtfsb1x	111111	crbD			00_000			0_0000			0000100110						Rc											
mtfsfx	111111	0	FM						0	B			1011000111						Rc									
mtfsfix	111111	crfD	00	00_000			IMM			0	0010000110						Rc											
mtmsr ²	011111	S			00_000			0_0000			0010010010						0											

Table A-3. Instructions by Mnemonic (Bin) (continued)

Name	0	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
mtspr ⁶	011111	S	spr						0111010011						0													
mtsr ²	011111	S	0	SR	0_0000						0011010010						0											
mtsrin ²	011111	S	00_000			B						0011110010						0										
mtvscr ³	000100	000_00	00_000			vB						11001000100						0										
mulhw _x	011111	D	A			B			0	001001011						Rc												
mulhw _x	011111	D	A			B			0	000001011						Rc												
mulli	000111	D	A			SIMM																						
mullw _x	011111	D	A			B			OE	011101011						Rc												
nand _x	011111	S	A			B			0111011100						Rc													
neg _x	011111	D	A			0_0000			OE	001101000						Rc												
nor _x	011111	S	A			B			0001111100						Rc													
or _x	011111	S	A			B			0110111100						Rc													
orc _x	011111	S	A			B			0110011100						Rc													
ori	011000	S	A			UIMM																						
oris	011001	S	A			UIMM																						
rfi ²	010011	000_00	00_000			0_0000			0000110010						0													
rlwim _x	010100	S	A			SH			MB	ME			Rc															
rlwin _m _x	010101	S	A			SH			MB	ME			Rc															
rlwn _m _x	010111	S	A			B			MB	ME			Rc															
sc	010001	000_0000_0000_0000_0000_0000_00																								1	0	
slw _x	011111	S	A			B			0000011000						Rc													
sraw _x	011111	S	A			B			1100011000						Rc													
sraw _i _x	011111	S	A			SH			1100011000						Rc													
srw _x	011111	S	A			B			1000011000						Rc													
stb	100110	S	A			d																						
stbu	100111	S	A			d																						
stb _x	011111	S	A			B			0011110111						0													
stb _x	011111	S	A			B			0011010111						0													
stfd	110110	S	A			d																						
stfd _u	110111	S	A			d																						
stfd _x	011111	S	A			B			1011110111						0													
stfd _x	011111	S	A			B			1011010111						0													
stfiw _x ¹	011111	S	A			B			1111010111						0													

Table A-3. Instructions by Mnemonic (Bin) (continued)

Name	0	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
stfs	110100				S					A																		
stfsu	110101				S					A																		
stfsux	011111				S					A				B														0
stfsx	011111				S					A				B														0
sth	101100				S					A																		
sthbrx	011111				S					A				B														0
sthv	101101				S					A																		
sthvx	011111				S					A				B														0
sthx	011111				S					A				B														0
stmw ⁵	101111				S					A																		
stswi ⁵	011111				S					A				NB														0
stswx ⁵	011111				S					A				B														0
stvebx ³	011111				vS					A				B														0
stvehx ³	011111				vS					A				B														0
stvewx ³	011111				vS					A				B														0
stvx ³	011111				vS					A				B														0
stvxi ³	011111				vS					A				B														0
stw	100100				S					A																		
stwbrx	011111				S					A				B														0
stwcx.	011111				S					A				B														1
stwu	100101				S					A																		
stwvx	011111				S					A				B														0
stwx	011111				S					A				B														0
subfx	011111				D					A				B			OE											Rc
subfcx	011111				D					A				B			OE											Rc
subfex	011111				D					A				B			OE											Rc
subfic	001000				D					A																		
subfmex	011111				D					A			0_0000				OE											Rc
subfzex	011111				D					A			0_0000				OE											Rc
sync	011111				000_00				00_000				0_0000															0
tlbia ⁴	011111				000_00				00_000				0_0000															0
tlbie ^{1,2}	011111				000_00				00_000				B															0
tlbld ^{1,2}	011111				000_00				00_000				B															0

Table A-3. Instructions by Mnemonic (Bin) (continued)

Name	0	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
tlbli ^{1,2}	011111	000_00					00_000			B											111110010							0
tlbsync ^{1,2}	011111	000_00					00_000			0_0000											1000110110							0
tw	011111	TO					A			B											0000000100							0
twi	000011	TO					A			SIMM																		
vaddcuw ³	000100	vD					vA			vB											0110000000							0
vaddfp ³	000100	vD					vA			vB											0000001010							0
vaddsbs ³	000100	vD					vA			vB											1100000000							0
vaddshs ³	000100	vD					vA			vB											1101000000							0
vaddsws ³	000100	vD					vA			vB											1110000000							0
vaddubm ³	000100	vD					vA			vB											0000000000							0
vaddubs ³	000100	vD					vA			vB											1000000000							0
vadduhm ³	000100	vD					vA			vB											0010000000							0
vadduhs ³	000100	vD					vA			vB											1001000000							0
vadduwm ³	000100	vD					vA			vB											0010000000							0
vadduws ³	000100	vD					vA			vB											1010000000							0
vand ³	000100	vD					vA			vB											1000000100							0
vandc ³	000100	vD					vA			vB											10001000100							0
vavgsb ³	000100	vD					vA			vB											10100000010							0
vavgsh ³	000100	vD					vA			vB											10101000010							0
vavgsw ³	000100	vD					vA			vB											10110000010							0
vavgub ³	000100	vD					vA			vB											10000000010							0
vavguh ³	000100	vD					vA			vB											10001000010							0
vavguw ³	000100	vD					vA			vB											10010000010							0
vcfsx ³	000100	vD					UIMM			vB											01101001010							
vcfux ³	000100	vD					UIMM			vB											1100001010							0
vcmpbfp ^{x3}	000100	vD					vA			vB				Rc							1111000110							
vcmpqfp ^{x3}	000100	vD					vA			vB				Rc							0011000110							
vcmpqub ^{x3}	000100	vD					vA			vB				Rc							0000000110							
vcmpquh ^{x3}	000100	vD					vA			vB				Rc							0001000110							
vcmpquw ^{x3}	000100	vD					vA			vB				Rc							0010000110							
vcmpgfp ^{x3}	000100	vD					vA			vB				Rc							0111000110							
vcmpgtfp ^{x3}	000100	vD					vA			vB				Rc							1011000110							
vcmpgtsb ^{x3}	000100	vD					vA			vB				Rc							1100000110							

Table A-3. Instructions by Mnemonic (Bin) (continued)

Name	0	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	
vcmpgtsh ^{x3}	000100		vD				vA						vB		Rc														1101000110
vcmpgtsw ^{x3}	000100		vD				vA						vB		Rc														1110000110
vcmpgtub ^{x3}	000100		vD				vA						vB		Rc														1000000110
vcmpgtuh ^{x3}	000100		vD				vA						vB		Rc														1001000110
vcmpgtuw ^{x3}	000100		vD				vA						vB		Rc														1010000110
vctxs ^{s3}	000100		vD						UIMM				vB																1111001010
vctux ^{s3}	000100		vD						UIMM				vB																1110001010
vexptfp ^{s3}	000100		vD						00_000				vB																110001010
vlogefp ^{s3}	000100		vD						00_000				vB																111001010
vmaddfp ^{s3}	000100		vD				vA						vB							vC								101110	
vmaxfp ^{s3}	000100		vD				vA						vB																10000001010
vmaxsb ^{s3}	000100		vD				vA						vB																0100000010
vmaxsh ^{s3}	000100		vD				vA						vB																0101000010
vmaxsw ^{s3}	000100		vD				vA						vB																0110000010
vmaxub ^{s3}	000100		vD				vA						vB																0000000010
vmaxuh ^{s3}	000100		vD				vA						vB																0001000010
vmaxuw ^{s3}	000100		vD				vA						vB																0010000010
vmhaddshs ^{s3}	000100		vD				vA						vB							vC									100000
vmhraddshs ^{s3}	000100		vD				vA						vB							vC									100001
vminfp ^{s3}	000100		vD				vA						vB																10001001010
vminsb ^{s3}	000100		vD				vA						vB																1100000010
vminsh ^{s3}	000100		vD				vA						vB																1101000010
vminsw ^{s3}	000100		vD				vA						vB																1110000010
vminub ^{s3}	000100		vD				vA						vB																1000000010
vminuh ^{s3}	000100		vD				vA						vB																1001000010
vminuw ^{s3}	000100		vD				vA						vB																1010000010
vmladduhm ^{s3}	000100		vD				vA						vB							vC									100010
vmrghb ^{s3}	000100		vD				vA						vB																0000001100
vmrghh ^{s3}	000100		vD				vA						vB																0001001100
vmrghw ^{s3}	000100		vD				vA						vB																0010001100
vmrglb ^{s3}	000100		vD				vA						vB																0100001100
vmrglh ^{s3}	000100		vD				vA						vB																0101001100
vmrglw ^{s3}	000100		vD				vA						vB																0110001100

Table A-3. Instructions by Mnemonic (Bin) (continued)

Name	0	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	
vmsummbm ³	000100		vD		vA		vB		vC																			100101	
vmsumshm ³	000100		vD		vA		vB		vC																				101000
vmsumshs ³	000100		vD		vA		vB		vC																				101001
vmsumubm ³	000100		vD		vA		vB		vC																				100100
vmsumuhm ³	000100		vD		vA		vB		vC																				100110
vmsumuhs ³	000100		vD		vA		vB		vC																				100111
vmulesb ³	000100		vD		vA		vB																						0100001000
vmulesh ³	000100		vD		vA		vB																						1101001000
vmuleub ³	000100		vD		vA		vB																						1000001000
vmuleuh ³	000100		vD		vA		vB																						1001001000
vmulosb ³	000100		vD		vA		vB																						0100001000
vmulosh ³	000100		vD		vA		vB																						0101001000
vmuloub ³	000100		vD		vA		vB																						0000001000
vmulouh ³	000100		vD		vA		vB																						0001001000
vnmsubfp ³	000100		vD		vA		vB		vC																				101111
vnor ³	000100		vD		vA		vB																						1010000100
vor ³	000100		vD		vA		vB																						10010000100
vperm ³	000100		vD		vA		vB		vC																				101011
vpkpx ³	000100		vD		vA		vB																						1100001110
vpkshs ³	000100		vD		vA		vB																						0110001110
vpkshus ³	000100		vD		vA		vB																						0100001110
vpkswss ³	000100		vD		vA		vB																						0111001110
vpkswus ³	000100		vD		vA		vB																						0101001110
vpkuhum ³	000100		vD		vA		vB																						0000001110
vpkuhus ³	000100		vD		vA		vB																						0010001110
vpkuwum ³	000100		vD		vA		vB																						0001001110
vpkuwus ³	000100		vD		vA		vB																						0011001110
vrefp ³	000100		vD		00_000		vB																						0100001010
vrfim ³	000100		vD		00_000		vB																						1011001010
vrfin ³	000100		vD		00_000		vB																						1000001010
vrfig ³	000100		vD		00_000		vB																						1010001010
vrfiz ³	000100		vD		00_000		vB																						1001001010
vrlb ³	000100		vD		vA		vB																						0000000100

Table A-3. Instructions by Mnemonic (Bin) (continued)

Name	0	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	
vrh ³	000100		vD							vA					vB														0001000100
vrlw ³	000100		vD							vA					vB														0010000100
vrsqrtefp ³	000100		vD							00_000					vB														0101001010
vse ³	000100		vD							vA					vB					vC									101010
vsi ³	000100		vD							vA					vB														0111000100
vslb ³	000100		vD							vA					vB														0100000100
vsldoi ³	000100		vD							vA					vB		0			SH									101100
vslh ³	000100		vD							vA					vB														0101000100
vslo ³	000100		vD							vA					vB														10000001100
vslw ³	000100		vD							vA					vB														0110000100
vspltb ³	000100		vD							UIMM					vB														1000001100
vsplth ³	000100		vD							UIMM					vB														1001001100
vspltisb ³	000100		vD							SIMM					0_0000														1100001100
vspltish ³	000100		vD							SIMM					0_0000														1101001100
vspltisw ³	000100		vD							SIMM					0_0000														1110001100
vspltw ³	000100		vD							UIMM					vB														1010001100
vsr ³	000100		vD							vA					vB														1011000100
vsrab ³	000100		vD							vA					vB														1100000100
vsrah ³	000100		vD							vA					vB														1101000100
vsraw ³	000100		vD							vA					vB														1110000100
vsrb ³	000100		vD							vA					vB														1000000100
vsrh ³	000100		vD							vA					vB														1001000100
vsro ³	000100		vD							vA					vB														10001001100
vsrw ³	000100		vD							vA					vB														1010000100
vsubcuw ³	000100		vD							vA					vB														10110000000
vsubfp ³	000100		vD							vA					vB														0001001010
vsubsb ³	000100		vD							vA					vB														11100000000
vsubsh ³	000100		vD							vA					vB														11101000000
vsubsw ³	000100		vD							vA					vB														11110000000
vsubbm ³	000100		vD							vA					vB														10000000000
vsububs ³	000100		vD							vA					vB														11000000000
vsubuhm ³	000100		vD							vA					vB														10001000000
vsubuhs ³	000100		vD							vA					vB														11001000000

Table A-3. Instructions by Mnemonic (Bin) (continued)

Name	0	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31		
vsubuwm ³	000100				vD				vA					vB															1001000000	
vsubuws ³	000100				vD				vA					vB															1101000000	
vsumsws ³	000100				vD				vA					vB															11110001000	
vsum2sws ³	000100				vD				vA					vB															11010001000	
vsum4sbs ³	000100				vD				vA					vB															11100001000	
vsum4shs ³	000100				vD				vA					vB															11001001000	
vsum4ubs ³	000100				vD				vA					vB															11000001000	
vupkhp ³	000100				vD				00_000					vB															1101001110	
vupkhsb ³	000100				vD				00_000					vB															1000001110	
vupksh ³	000100				vD				00_000					vB															1001001110	
vupklp ³	000100				vD				00_000					vB															1111001110	
vupklb ³	000100				vD				00_000					vB															1010001110	
vupklsh ³	000100				vD				00_000					vB															1011001110	
vxor ³	000100				vD				vA					vB															10011000100	
xorx	011111				S				A					B															0100111100	Rc
xori	011010				S				A																				UIMM	
xoris	011011				S				A																				UIMM	

- ¹ Optional to the PowerPC architecture but implemented by the e600 core.
- ² Supervisor-level instruction.
- ³ AltiVec technology-specific instruction.
- ⁴ Optional instruction not implemented by the e600 core.
- ⁵ Load/store string/multiple instruction.
- ⁶ Supervisor- and user-level instruction.

A.4 Instructions Sorted by Opcode (Binary)

Table A-4 lists the instructions implemented in the e600 core in binary numerical order by opcode.

Key:

Reserved bits

Table A-4. Instructions by Primary and Secondary Opcode (Bin)

Name	0	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31		
twi	000011				TO				A																				SIMM	
vaddubm ¹	000100				vD				vA					vB															0000000000	0
vmaxub ¹	000100				vD				vA					vB															0000000010	
vrlb ¹	000100				vD				vA					vB															0000000100	

Table A-4. Instructions by Primary and Secondary Opcode (Bin) (continued)

Name	0	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31							
vcmpesubx ¹	000100		vD		vA		vB		Rc	000000110																									
vmuloub ¹	000100		vD		vA		vB			0000001000																									
vaddfp ¹	000100		vD		vA		vB			0000001010					0																				
vmrghb ¹	000100		vD		vA		vB			0000001100																									
vpkuhum ¹	000100		vD		vA		vB			0000001110																									
vmhaddshs ¹	000100		vD		vA		vB		vC	100000																									
vmhraddshs ¹	000100		vD		vA		vB		vC	100001																									
vmladduhm ¹	000100		vD		vA		vB		vC	100010																									
vmsumubm ¹	000100		vD		vA		vB		vC	100100																									
vmsummbm ¹	000100		vD		vA		vB		vC	100101																									
vmsumuhm ¹	000100		vD		vA		vB		vC	100110																									
vmsumuhs ¹	000100		vD		vA		vB		vC	100111																									
vmsumshm ¹	000100		vD		vA		vB		vC	101000																									
vmsumshs ¹	000100		vD		vA		vB		vC	101001																									
vsel ¹	000100		vD		vA		vB		vC	101010																									
vperm ¹	000100		vD		vA		vB		vC	101011																									
vsldoi ¹	000100		vD		vA		vB	0	SH	101100																									
vmaddfp ¹	000100		vD		vA		vB		vC	101110																									
vnmsubfp ¹	000100		vD		vA		vB		vC	101111																									
vadduhm ¹	000100		vD		vA		vB			001000000					0																				
vmaxuh ¹	000100		vD		vA		vB			0001000010																									
vrh ¹	000100		vD		vA		vB			0001000100																									
vcmpesquh ¹	000100		vD		vA		vB		Rc	0001000110																									
vmulouh ¹	000100		vD		vA		vB			0001001000																									
vsubfp ¹	000100		vD		vA		vB			0001001010																									
vmrghh ¹	000100		vD		vA		vB			0001001100																									
vpkuwum ¹	000100		vD		vA		vB			0001001110																									
vadduwm ¹	000100		vD		vA		vB			0010000000					0																				
vmaxuw ¹	000100		vD		vA		vB			0010000010																									
vrlw ¹	000100		vD		vA		vB			0010000100																									
vcmpesquw ¹	000100		vD		vA		vB		Rc	0010000110																									
vmrghw ¹	000100		vD		vA		vB			0010001100																									
vpkuhus ¹	000100		vD		vA		vB			0010001110																									

Table A-4. Instructions by Primary and Secondary Opcode (Bin) (continued)

Name	0	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
vcmpeqfp ^{x1}	000100	vD	vA	vB	Rc	0011000110																						
vpkuwus ¹	000100	vD	vA	vB	0011001110																							
vmaxsb ¹	000100	vD	vA	vB	0100000010																							
vslb ¹	000100	vD	vA	vB	0100000100																							
vmulosb ¹	000100	vD	vA	vB	0100001000																							
vrefp ¹	000100	vD	00_000	vB	0100001010																							
vmrglb ¹	000100	vD	vA	vB	0100001100																							
vpkshus ¹	000100	vD	vA	vB	0100001110																							
vmaxsh ¹	000100	vD	vA	vB	0101000010																							
vslh ¹	000100	vD	vA	vB	0101000100																							
vmulosh ¹	000100	vD	vA	vB	0101001000																							
vrsqrtefp ¹	000100	vD	00_000	vB	0101001010																							
vmrglh ¹	000100	vD	vA	vB	0101001100																							
vpkswus ¹	000100	vD	vA	vB	0101001110																							
vaddcuw ¹	000100	vD	vA	vB	0110000000																				0			
vmaxsw ¹	000100	vD	vA	vB	0110000010																							
vslw ¹	000100	vD	vA	vB	0110000100																							
vexptefp ¹	000100	vD	00_000	vB	110001010																							
vmrglw ¹	000100	vD	vA	vB	0110001100																							
vpkshss ¹	000100	vD	vA	vB	0110001110																							
vsl ¹	000100	vD	vA	vB	0111000100																							
vcmpgefp ^{x1}	000100	vD	vA	vB	Rc	0111000110																						
vlogefp ¹	000100	vD	00_000	vB	111001010																							
vpkswss ¹	000100	vD	vA	vB	0111001110																							
vaddubs ¹	000100	vD	vA	vB	1000000000																				0			
vminub ¹	000100	vD	vA	vB	1000000010																							
vsrb ¹	000100	vD	vA	vB	1000000100																							
vcmpgtub ^{x1}	000100	vD	vA	vB	Rc	1000000110																						
vmuleub ¹	000100	vD	vA	vB	1000001000																							
vrfin ¹	000100	vD	00_000	vB	1000001010																							
vspltb ¹	000100	vD	UIMM	vB	1000001100																							
vupkhsb ¹	000100	D	00_000	B	1000001110																							
vadduhs ¹	000100	vD	vA	vB	1001000000																				0			

Table A-4. Instructions by Primary and Secondary Opcode (Bin) (continued)

Name	0	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	
vminuh ¹	000100		vD				vA				vB																		
vsrh ¹	000100		vD				vA				vB																		
vcmpgtuhx ¹	000100		vD				vA				vB							Rc											
vmuleuh ¹	000100		vD				vA				vB																		
vrfiz ¹	000100		vD				00_000				vB																		
vsplth ¹	000100		vD				UIMM				vB																		
vupksh ¹	000100		D				00_000				B																		
vadduws ¹	000100		vD				vA				vB																	0	
vminuw ¹	000100		vD				vA				vB																		
vsrw ¹	000100		vD				vA				vB																		
vcmpgtuwx ¹	000100		vD				vA				vB							Rc											
vrfig ¹	000100		vD				00_000				vB																		
vspltw ¹	000100		vD				UIMM				vB																		
vupklsb ¹	000100		D				00_000				B																		
vsr ¹	000100		vD				vA				vB																		
vcmpgtfpx ¹	000100		vD				vA				vB							Rc											
vrfig ¹	000100		vD				00_000				vB																		
vupklsh ¹	000100		D				00_000				B																		
vaddsb ¹	000100		vD				vA				vB																	0	
vminsb ¹	000100		vD				vA				vB																		
vsrab ¹	000100		vD				vA				vB																		
vcmpgtsbx ¹	000100		vD				vA				vB							Rc											
vmulesb ¹	000100		vD				vA				vB																		
vcfux ¹	000100		vD				UIMM				vB																	0	
vspltisb ¹	000100		vD				SIMM				0_0000																		
vpkpx ¹	000100		vD				vA				vB																		
vaddsh ¹	000100		vD				vA				vB																	0	
vminsh ¹	000100		vD				vA				vB																		
vsrah ¹	000100		vD				vA				vB																		
vcmpgtshx ¹	000100		vD				vA				vB							Rc											
vmulesh ¹	000100		vD				vA				vB																		
vcfsx ¹	000100		vD				UIMM				vB																		
vspltish ¹	000100		vD				SIMM				0_0000																		

Table A-4. Instructions by Primary and Secondary Opcode (Bin) (continued)

Name	0	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
vupkhp ¹	000100		vD				00_000		vB												1101001110							
vaddsw ¹	000100		vD				vA		vB												1110000000							0
vminsw ¹	000100		vD				vA		vB												1110000010							
vsraw ¹	000100		vD				vA		vB												1110000100							
vcmpgtsw ¹	000100		vD				vA		vB	Rc											1110000110							
vctux ¹	000100		vD				UIMM		vB												1110001010							
vspltisw ¹	000100		vD				SIMM		0_0000												1110001100							
vcmpbfp ¹	000100		vD				vA		vB	Rc											1111000110							
vctxs ¹	000100		vD				UIMM		vB												1111001010							
vupklpx ¹	000100		vD				00_000		vB												1111001110							
vsububm ¹	000100		vD				vA		vB												1000000000							
vavgub ¹	000100		vD				vA		vB												1000000010							0
vand ¹	000100		vD				vA		vB												1000000100							0
vmaxfp ¹	000100		vD				vA		vB												10000001010							
vslo ¹	000100		vD				vA		vB												10000001100							
vsubuhm ¹	000100		vD				vA		vB												10001000000							
vavguh ¹	000100		vD				vA		vB												10001000010							0
vandc ¹	000100		vD				vA		vB												10001000100							0
vminfp ¹	000100		vD				vA		vB												10001001010							
vsro ¹	000100		vD				vA		vB												10001001100							
vsubuwm ¹	000100		vD				vA		vB												10010000000							
vavguw ¹	000100		vD				vA		vB												10010000010							0
vor ¹	000100		vD				vA		vB												10010000100							
vxor ¹	000100		vD				vA		vB												10011000100							
vavgsb ¹	000100		vD				vA		vB												10100000010							0
vnor ¹	000100		vD				vA		vB												10100000100							
vavgsh ¹	000100		vD				vA		vB												10101000010							0
vsubcuw ¹	000100		vD				vA		vB												10110000000							
vavgsw ¹	000100		vD				vA		vB												10110000010							0
vsububs ¹	000100		vD				vA		vB												11000000000							
mfvscr ¹	000100		vD				00_000		0_0000												11000000100							0
vsum4ubs ¹	000100		vD				vA		vB												11000001000							
vsubuhs ¹	000100		vD				vA		vB												11001000000							

Table A-4. Instructions by Primary and Secondary Opcode (Bin) (continued)

Name	0	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
mtvscr ¹	000100	000_00			00_000			vB			11001000100						0											
vsum4shs ¹	000100	vD			vA			vB			11001001000																	
vsubuws ¹	000100	vD			vA			vB			11010000000																	
vsum2sws ¹	000100	vD			vA			vB			11010001000																	
vsubsbs ¹	000100	vD			vA			vB			11100000000																	
vsum4sbs ¹	000100	vD			vA			vB			11100001000																	
vsubshs ¹	000100	vD			vA			vB			11101000000																	
vsubsws ¹	000100	vD			vA			vB			11110000000																	
vsumsws ¹	000100	vD			vA			vB			11110001000																	
mulld	000111	D			A			SIMM																				
subfic	001000	D			A			SIMM																				
cmpli	001010	crfD	0	L	A			UIMM																				
cmpi	001011	crfD	0	L	A			SIMM																				
addic	001100	D			A			SIMM																				
addic.	001101	D			A			SIMM																				
addi	001110	D			A			SIMM																				
addis	001111	D			A			SIMM																				
bcx	010000	BO			BI			BD						AA	LK													
sc	010001	000_0000_0000_0000_0000_00																								1	0	
bx	010010	LI																								AA	LK	
mcrf	010011	crfD	00	crfS	00	0_0000			00000000						0													
bclrx	010011	BO			BI			0_0000			0000010000						LK											
crnor	010011	crbD			crbA			crbB			0000100001						0											
rfti ²	010011	000_00			00_000			0_0000			0000110010						0											
crandc	010011	crbD			crbA			crbB			0010000001						0											
isync	010011	000_00			00_000			0_0000			0010010110						0											
crxor	010011	crbD			crbA			crbB			0011000001						0											
crnand	010011	crbD			crbA			crbB			0011100001						0											
crand	010011	crbD			crbA			crbB			0100000001						0											
creqv	010011	crbD			crbA			crbB			0100100001						0											
crorc	010011	crbD			crbA			crbB			0110100001						0											
cror	010011	crbD			crbA			crbB			0111000001						0											
bcctrx	010011	BO			BI			0_0000			1000010000						LK											

Table A-4. Instructions by Primary and Secondary Opcode (Bin) (continued)

Name	0	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
rlwimix	010100				S					A				SH							MB				ME			Rc
rlwinmx	010101				S					A				SH							MB				ME			Rc
rlwnmx	010111				S					A				B							MB				ME			Rc
ori	011000				S					A				UIMM														
oris	011001				S					A				UIMM														
xori	011010				S					A				UIMM														
xoris	011011				S					A				UIMM														
andi.	011100				S					A				UIMM														
andis.	011101				S					A				UIMM														
cmp	011111	crfD		0	L					A				B							000000000					0		
tw	011111				TO					A				B							0000000100					0		
lvs1 ¹	011111				vD					A				B							0000000110					0		
lvebx ¹	011111				vD					A				B							0000000111					0		
subfcx	011111				D					A				B		OE					000001000					Rc		
addcx	011111				D					A				B		OE					000001010					Rc		
mulhwux	011111				D					A				B		0					000001011					Rc		
mfcx	011111				D					00_000				0_0000							0000010011					0		
lwarx	011111				D					A				B							0000010100					0		
lwzx	011111				D					A				B							0000010111					0		
slwx	011111				S					A				B							0000011000					Rc		
cntlzwx	011111				S					A				0_0000							0000011010					Rc		
andx	011111				S					A				B							000011100					Rc		
cmpl	011111	crfD		0	L					A				B							0000100000					0		
lvsr ¹	011111				vD					A				B							0000100110					0		
lvehx ¹	011111				vD					A				B							0000100111					0		
subfx	011111				D					A				B		OE					000101000					Rc		
dcbst	011111				000_00					A				B							0000110110					0		
lwzux	011111				D					A				B							0000110111					0		
andcx	011111				S					A				B							000111100					Rc		
lvewx ¹	011111				vD					A				B							0001000111					0		
mulhwx	011111				D					A				B		0					001001011					Rc		
mfmsr ²	011111				D					00_000				0_0000							0001010011					0		
dcbf	011111				000_00					A				B							0001010110					0		

Table A-4. Instructions by Primary and Secondary Opcode (Bin) (continued)

Name	0	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	
lbzx	011111				D					A					B														0
lvx ¹	011111				vD					A					B														0
negx	011111				D					A			0_0000		OE														Rc
lbzux	011111				D					A					B														0
norx	011111				S					A					B														Rc
stvebx ¹	011111				vS					A					B														0
subfex	011111				D					A					B			OE											Rc
addex	011111				D					A					B			OE											Rc
mtcrf	011111				S		0					CRM					0												0
mtmsr ²	011111				S			00_000					0_0000																0
stwcx.	011111				S					A					B														1
stwx	011111				S					A					B														0
stvehx ¹	011111				vS					A					B														0
stwux	011111				S					A					B														0
stvewx ¹	011111				vS					A					B														0
subfzex	011111				D					A			0_0000		OE														Rc
addzex	011111				D					A			0_0000		OE														Rc
mtsr ²	011111				S		0			SR			0_0000																0
stbx	011111				S					A					B														0
stvx ¹	011111				vS					A					B														0
subfmex	011111				D					A			0_0000		OE														Rc
addmex	011111				D					A			0_0000		OE														Rc
mullwx	011111				D					A					B			OE											Rc
mtsrin ²	011111				S			00_000							B														0
dcbtst	011111					000_00				A					B														0
stbux	011111				S					A					B														0
addx	011111				D					A					B			OE											Rc
dcbt	011111					000_00				A					B														0
lhzx	011111				D					A					B														0
eqvx	011111				S					A					B														Rc
tlbie ^{2,3}	011111					000_00		00_000							B														0
eciwx ³	011111				D					A					B														0
lhzux	011111				D					A					B														0

Table A-4. Instructions by Primary and Secondary Opcode (Bin) (continued)

Name	0	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
xorx	011111	S			A			B			0100111100						Rc											
mfspir ⁴	011111	D			spr						0101010011						0											
dst ¹	011111	T	00	STRM	A			B			0101010110						0											
dstt ¹	011111	1	00	STRM	A			B			0101010110						0											
lhax	011111	D			A			B			0101010111						0											
lvxl ¹	011111	vD			A			B			0101100111						0											
tlbia ⁵	011111	000_00			00_000			0_0000			0101110010						0											
mftb	011111	D			tbr						0101110011						0											
dstst ¹	011111	T	00	STRM	A			B			0101110110						0											
dststt ¹	011111	1	00	STRM	A			B			0101110110						0											
lhax	011111	D			A			B			0101110111						0											
sthx	011111	S			A			B			110010111						0											
orcx	011111	S			A			B			0110011100						Rc											
ecowx ³	011111	S			A			B			0110110110						0											
sthux	011111	S			A			B			110110111						0											
orx	011111	S			A			B			0110111100						Rc											
divwux	011111	D			A			B			OE	1 1100 1011					Rc											
mtspr ⁴	011111	S			spr						0111010011						0											
dcbi ²	011111	000_00			A			B			0111010110						0											
nandx	011111	S			A			B			0111011100						Rc											
stvxl ¹	011111	vS			A			B			0111100111						0											
divwx	011111	D			A			B			OE	1 1110 1011					Rc											
mcrxr	011111	crfD	00		00_000			0_0000			1000000000						0											
lswx ⁶	011111	D			A			B			1000010101						0											
lwbrx	011111	D			A			B			1000010110						0											
lfsx	011111	D			A			B			1000010111						0											
srwx	011111	S			A			B			1000011000						Rc											
tlbsync ^{2,3}	011111	000_00			00_000			0_0000			1000110110						0											
lfsux	011111	D			A			B			1000110111						0											
mfsr ²	011111	D			0	SR		0_0000			1001010011						0											
lswi ⁶	011111	D			A			NB			1001010101						0											
sync	011111	000_00			00_000			0_0000			1001010110						0											
lfdx	011111	D			A			B			1001010111						0											

Table A-4. Instructions by Primary and Secondary Opcode (Bin) (continued)

Name	0	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	
lfdx	011111				D					A					B														0
mfsrin ²	011111				D					00_000					B														0
stswx ⁶	011111				S					A					B														0
stwbrx	011111				S					A					B														0
stfsx	011111				S					A					B														0
stfsux	011111				S					A					B														0
stswi ⁶	011111				S					A					NB														0
stfdx	011111				S					A					B														0
dcba ³	011111									000_00					A														0
stfdx	011111				S					A					B														0
lhbrx	011111				D					A					B														0
srawx	011111				S					A					B														Rc
dss ¹	011111	A	00	STRM						00_000					0_0000														0
dssall ¹	011111	A	00	STRM						00_000					0_0000														0
srawix	011111				S					A					SH														Rc
eieio	011111									000_00					00_000														0
sthbrx	011111				S					A					B														0
extshx	011111				S					A					0_0000														Rc
extsbx	011111				S					A					0_0000														Rc
tlbld ^{2,3}	011111									000_00					00_000														0
icbi	011111									000_00					A														0
stfiwx ³	011111				S					A					B														0
tlbli ^{2,3}	011111									000_00					00_000														0
dcbz	011111									000_00					A														0
lwz	100000				D					A																			d
lwzu	100001				D					A																			d
lbz	100010				D					A																			d
lbzu	100011				D					A																			d
stw	100100				S					A																			d
stwu	100101				S					A																			d
stb	100110				S					A																			d
stbu	100111				S					A																			d
lhz	101000				D					A																			d

Table A-4. Instructions by Primary and Secondary Opcode (Bin) (continued)

Name	0	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	
lhzu	101001				D					A																			d
lha	101010				D					A																			d
lhau	101011				D					A																			d
sth	101100				S					A																			d
sthu	101101				S					A																			d
lmw ⁶	101110				D					A																			d
stmw ⁶	101111				S					A																			d
lfs	110000				D					A																			d
lfsu	110001				D					A																			d
lfd	110010				D					A																			d
lfdx	110011				D					A																			d
stfs	110100				S					A																			d
stfsu	110101				S					A																			d
stfd	110110				S					A																			d
stfdx	110111				S					A																			d
fdivsx	111011				D					A				B				0000_0						1 0010				Rc	
fsubsx	111011				D					A				B				0000_0						1 0100				Rc	
faddsx	111011				D					A				B				0000_0						1 0101				Rc	
fsqrtx ⁵	111011				D					00_000				B				0000_0						1 0110				Rc	
fresx ³	111011				D					00_000				B				0000_0						1 1000				Rc	
fmulx	111011				D					A				0_0000				C						1 1001				Rc	
fmsubx	111011				D					A				B				C						1 1100				Rc	
fmadx	111011				D					A				B				C						1 1101				Rc	
fnmsubx	111011				D					A				B				C						1 1110				Rc	
fnmadx	111011				D					A				B				C						1 1111				Rc	
fcmpu	111111			crfD			00			A				B										0000000000				0	
frspx	111111				D					00_000				B										0000001100				Rc	
fctiw	111111				D					00_000				B										0000001110				Rc	
fctiwz	111111				D					00_000				B										0000001111				Rc	
fdivx	111111				D					A				B				0000_0						1 0010				Rc	
fsubx	111111				D					A				B				0000_0						1 0100				Rc	
faddx	111111				D					A				B				0000_0						1 0101				Rc	
fsqrtx ⁵	111111				D					00_000				B				0000_0						1 0110				Rc	

Table A-4. Instructions by Primary and Secondary Opcode (Bin) (continued)

Name	0	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
fselx ³	111111				D				A				B						C				1	0	1	1	1	Rc
fmulx	111111				D				A				0_0000						C				1	1	0	0	1	Rc
frsqrtox ³	111111				D				00_000				B						0000_0				1	1	0	1	0	Rc
fmsubx	111111				D				A				B						C				1	1	1	0	0	Rc
fmaddx	111111				D				A				B						C				1	1	1	0	1	Rc
fnmsubx	111111				D				A				B						C				1	1	1	0	0	Rc
fnmaddx	111111				D				A				B						C				1	1	1	1	1	Rc
fcmpo	111111			crfD		00			A				B						0000100000					0				
mtfsb1x	111111			crbD				00_000				0_0000							0000100110					Rc				
fnegx	111111				D				00_000				B						0000101000					Rc				
mcrfs	111111			crfD		00		crfS		00			0_0000						001000000					0				
mtfsb0x	111111			crbD				00_000				0_0000							0001000110					Rc				
fmrx	111111				D				00_000				B						0001001000					Rc				
mtfsfix	111111			crfD		00		00_000				IMM		0					0010000110					Rc				
fnabsx	111111				D				00_000				B						0010001000					Rc				
fabsx	111111				D				00_000				B						0100001000					Rc				
mffsx	111111				D				00_000				0_0000						1001000111					Rc				
mtfsfx	111111		0					FM				0			B				1011000111					Rc				

- ¹ AltiVec technology-specific instruction.
- ² Supervisor-level instruction.
- ³ Optional to the PowerPC architecture but implemented by the e600 core.
- ⁴ Supervisor- and user-level instruction.
- ⁵ Optional instruction not implemented by the e600 core.
- ⁶ Load/store string/multiple instruction.

A.5 Instructions Grouped by Functional Categories

Table A-5 through Table A-44 list the e600 core instructions grouped by function.

Key: Reserved bits

Table A-5. Integer Arithmetic Instructions

Name	0	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
addx		31				D			A				B					OE					266					Rc
addcx		31				D			A				B					OE					10					Rc
addex		31				D			A				B					OE					138					Rc

Table A-5. Integer Arithmetic Instructions (continued)

addi	14	D	A	SIMM			
addic	12	D	A	SIMM			
addic.	13	D	A	SIMM			
addis	15	D	A	SIMM			
addmex	31	D	A	0 0 0 0 0	OE	234	Rc
addzex	31	D	A	0 0 0 0 0	OE	202	Rc
divwx	31	D	A	B	OE	491	Rc
divwux	31	D	A	B	OE	459	Rc
mulhwx	31	D	A	B	0	75	Rc
mulhwux	31	D	A	B	0	11	Rc
mulli	07	D	A	SIMM			
mullwx	31	D	A	B	OE	235	Rc
negx	31	D	A	0 0 0 0 0	OE	104	Rc
subfx	31	D	A	B	OE	40	Rc
subfcx	31	D	A	B	OE	8	Rc
subfic	08	D	A	SIMM			
subfex	31	D	A	B	OE	136	Rc
subfmex	31	D	A	0 0 0 0 0	OE	232	Rc
subfzex	31	D	A	0 0 0 0 0	OE	200	Rc

Table A-6. Integer Compare Instructions

Name	0	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
cmp	31	crfD	0	L	A	B	0 0 0 0 0 0 0 0 0 0										0											
cmpi	11	crfD	0	L	A	SIMM																						
cmpl	31	crfD	0	L	A	B	32										0											
cmpli	10	crfD	0	L	A	UIMM																						

Table A-7. Integer Logical Instructions

Name	0	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
andx	31	S	A	B	28										Rc													
andcx	31	S	A	B	60										Rc													
andi.	28	S	A	UIMM																								
andis.	29	S	A	UIMM																								
cntlzwx	31	S	A	0 0 0 0 0	26										Rc													

Table A-7. Integer Logical Instructions (continued)

eqvx	31	S	A	B	284	Rc
extsbx	31	S	A	0 0 0 0 0	954	Rc
extshx	31	S	A	0 0 0 0 0	922	Rc
nandx	31	S	A	B	476	Rc
norx	31	S	A	B	124	Rc
orx	31	S	A	B	444	Rc
orcx	31	S	A	B	412	Rc
ori	24	S	A	UIMM		
oris	25	S	A	UIMM		
vand ¹	04	vD	vA	vB	1028	0
vandc ¹	04	vD	vA	vB	1092	0
vnor ¹	04	vD	vA	vB	1284	
vor ¹	04	vD	vA	vB	1156	
vxor ¹	04	D	A	B	1220	
xorx	31	S	A	B	316	Rc
xori	26	S	A	UIMM		
xoris	27	S	A	UIMM		

¹ AltiVec technology-specific instruction.

Table A-8. Integer Rotate Instructions

Name	0	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	
rlwimx	22	S								A									SH		MB								Rc
rlwinmx	20	S								A									SH		MB								Rc
rlwnmx	21	S								A									SH		MB								Rc

Table A-9. Integer Shift Instruction

Name	0	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	
slwx	31	S								A									B			24							Rc
srawx	31	S								A									B			792							Rc
srawix	31	S								A									SH			824							Rc
srwx	31	S								A									B			536							Rc

Table A-10. Floating-Point Arithmetic Instructions

Name	0	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	
faddx	63				D					A					B				0	0	0	0		21				Rc	
faddsx	59				D					A					B				0	0	0	0		21				Rc	
fdivx	63				D					A					B				0	0	0	0		18				Rc	
fdivsx	59				D					A					B				0	0	0	0		18				Rc	
fmulx	63				D					A			0	0	0	0	0			C			25					Rc	
fmulsx	59				D					A			0	0	0	0	0			C			25					Rc	
fresx ¹	59				D			0	0	0	0	0			B			0	0	0	0		24					Rc	
frsqrtox ¹	63				D			0	0	0	0	0			B			0	0	0	0		26					Rc	
fsubx	63				D					A					B				0	0	0	0		20				Rc	
fsubsx	59				D					A					B				0	0	0	0		20				Rc	
fselx	63				D					A					B					C			23					Rc	
fsqrtx ²	63				D			0	0	0	0	0			B			0	0	0	0		22					Rc	
fsqrtsx ²	59				D			0	0	0	0	0			B			0	0	0	0		22					Rc	
vaddfp ³	04				vD					vA					vB							10						0	
vmaxfp ³	04				vD					vA					vB								1034						
vminfp ³	04				vD					vA					vB								1098						
vsubfp ³	04				vD					vA					vB								74						

¹ Optional to the PowerPC architecture but implemented by the e600 core.

² Optional instruction not implemented by the e600 core.

³ AltiVec technology-specific instruction.

Table A-11. Floating-Point Multiply-Add Instructions

Name	0	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
fmaddx	63				D					A					B					C				29				Rc
fmaddsx	59				D					A					B					C				29				Rc
fmsubx	63				D					A					B					C				28				Rc
fmsubsx	59				D					A					B					C				28				Rc
fnmaddx	63				D					A					B					C				31				Rc
fnmaddsx	59				D					A					B					C				31				Rc
fnmsubx	63				D					A					B					C				30				Rc
fnmsubsx	59				D					A					B					C				30				Rc
vmaddfp ¹	04				vD					vA					vB					vC				46				
vnmsubfp ¹	04				vD					vA					vB					vC				47				

¹ AltiVec technology-specific instruction.

Table A-12. Floating-Point Rounding and Conversion Instructions

Name	0	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
fctiw_x	63				D				0	0	0	0		B								14						Rc
fctiw_z	63				D				0	0	0	0		B								15						Rc
frsp_x	63				D				0	0	0	0		B								12						Rc
vcfs_x ¹	04				vD				U	I	M	M		vB								842						
vcfux ¹	04				vD				U	I	M	M		vB								778					0	
vctsx_s ¹	04				vD				U	I	M	M		vB								970						
vctux_s ¹	04				vD				U	I	M	M		vB								906						
vrfim ¹	04				vD				0	0	0	0		vB								714						
vrfin ¹	04				vD				0	0	0	0		vB								522						
vrfig ¹	04				vD				0	0	0	0		vB								650						
vrfiz ¹	04				vD				0	0	0	0		vB								586						

¹ AltiVec technology-specific instruction.

Table A-13. Floating-Point Compare Instructions

Name	0	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
fcmpo	63				crfD				0	0		A		B								32						0
fcmpu	63				crfD				0	0		A		B								0						0
vcmpbfp_x ¹	04				vD				v	A		v	B	Rc								966						
vcmpqfp_x ¹	04				vD				v	A		v	B	Rc								198						
vcmpgefp_x ¹	04				vD				v	A		v	B	Rc								454						
vcmpgtfp_x ¹	04				vD				v	A		v	B	Rc								710						

¹ AltiVec technology-specific instruction.

Table A-14. Floating-Point Status and Control Register Instructions

Name	0	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
mcrfs	63				crfD				0	0		crfS		0	0		0	0	0	0	0		64					0
mffs_x	63				D				0	0	0	0	0	0	0	0	0	0	0	0		583						Rc
mtfsb0_x	63				crbD				0	0	0	0	0	0	0	0	0	0	0		70							Rc
mtfsb1_x	63				crbD				0	0	0	0	0	0	0	0	0	0	0		38							Rc
mtfsf_x	31		0												B							711						Rc
mtfsfix	63				crfD				0	0	0	0	0	0	0	0	0	0	0		134							Rc

Table A-15. Integer Load Instructions

Name	0	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	
lbz	34				D					A																			
lbzu	35				D					A																			
lbzux	31				D					A				B								119						0	
lbzx	31				D					A				B								87						0	
lha	42				D					A																			
lhau	43				D					A																			
lhaux	31				D					A				B								375						0	
lhax	31				D					A				B								343						0	
lhz	40				D					A																			
lhzu	41				D					A																			
lhzux	31				D					A				B								311						0	
lhzx	31				D					A				B								279						0	
lvebx ¹	31				vD					A				B								7						0	
lvehx ¹	31				vD					A				B								39						0	
lviewx ¹	31				vD					A				B								71						0	
lvx ¹	31				vD					A				B								103						0	
lvxl ¹	31				vD					A				B								359						0	
lwz	32				D					A																			
lwzu	33				D					A																			
lwzux	31				D					A				B								55						0	
lwzx	31				D					A				B								23						0	

¹ AltiVec technology-specific instruction.

Table A-16. Integer Store Instructions

Name	0	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	
stb	38				S					A																			
stbu	39				S					A																			
stbux	31				S					A				B								247						0	
stbx	31				S					A				B								215						0	
sth	44				S					A																			
sthu	45				S					A																			
sthux	31				S					A				B								439						0	
sthx	31				S					A				B								407						0	

Table A-16. Integer Store Instructions (continued)

stw	36	S	A	d		
stwu	37	S	A	d		
stwu	31	S	A	B	183	0
stwx	31	S	A	B	151	0

Table A-17. Integer Load and Store with Byte Reverse Instructions

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

lhbrx	31	D	A	B	790	0
lwbrx	31	D	A	B	534	0
sthbrx	31	S	A	B	918	0
stwbrx	31	S	A	B	662	0

Table A-18. Integer Load and Store Multiple Instructions

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

lmw ¹	46	D	A	d		
stmw ¹	47	S	A	d		

¹ Load/store string/multiple instruction.

Table A-19. Integer Load and Store String Instructions

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

lswi ¹	31	D	A	NB	597	0
lswx ¹	31	D	A	B	533	0
stswi ¹	31	S	A	NB	725	0
stswx ¹	31	S	A	B	661	0

¹ Load/store string/multiple instruction.

Table A-20. Memory Synchronization Instructions

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

eieio	31	0 0 0 0 0	0 0 0 0 0	0 0 0 0 0	854	0
isync	19	0 0 0 0 0	0 0 0 0 0	0 0 0 0 0	150	0
lwarx	31	D	A	B	20	0
stwcx.	31	S	A	B	150	1
sync	31	0 0 0 0 0	0 0 0 0 0	0 0 0 0 0	598	0

Table A-21. Floating-Point Load Instructions

Name	0	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
lfd	50				D					A																		
lfd	51				D					A																		
lfdx	31				D					A					B													0
lfdx	31				D					A					B													0
lfs	48				D					A																		
lfsu	49				D					A																		
lfsux	31				D					A					B													0
lfsx	31				D					A					B													0

Table A-22. Floating-Point Store Instructions

Name	0	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
stfd	54				S					A																		
stfd	55				S					A																		
stfdx	31				S					A					B													0
stfdx	31				S					A					B													0
stfiwx¹	31				S					A					B													0
stfs	52				S					A																		
stfsu	53				S					A																		
stfsux	31				S					A					B													0
stfsx	31				S					A					B													0

¹ Optional to the PowerPC architecture but implemented by the e600 core.

Table A-23. Floating-Point Move Instructions

Name	0	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
fabsx	63				D					0	0	0	0		B													Rc
fmr_x	63				D					0	0	0	0		B													Rc
fnabs_x	63				D					0	0	0	0		B													Rc
fneg_x	63				D					0	0	0	0		B													Rc

Table A-24. Branch Instructions

Name	0	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	
bx	18																											AA	LK
bc_x	16					BO						BI																AA	LK

Table A-28. Processor Control Instructions (continued)

mtmsr ¹	31	S	00000	00000	146	0
mtspr ²	31	D	spr		467	0

¹ Supervisor-level instruction

² Supervisor- and user-level instruction

Table A-29. Cache Management Instructions

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

dcba ¹	31	00000	A	B	758	0
dcbf	31	00000	A	B	86	0
dcbi ²	31	00000	A	B	470	0
dcbst	31	00000	A	B	54	0
dcbt	31	00000	A	B	278	0
dcbtst	31	00000	A	B	246	0
dcbz	31	00000	A	B	1014	0
icbi	31	00000	A	B	982	0

¹ Optional to the PowerPC but implemented by the e600 core.

² Supervisor-level instruction.

Table A-30. Segment Register Manipulation Instructions

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

mfsr ¹	31	D	0	SR	00000	595	0
mfsrin ¹	31	D	00000		B	659	0
mtsr ¹	31	S	0	SR	00000	210	0
mtsrin ¹	31	S	00000		B	242	0

¹ Supervisor-level instruction

Table A-31. Lookaside Buffer Management Instructions

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

tlbia ¹	31	00000	00000	00000	370	0
tlbie ^{2,3}	31	00000	00000	B	306	0
tlbld ^{2,3}	31	00000	00000	B	978	0
tlbli ^{2,3}	31	00000	00000	B	1010	0
tlbsync ^{2,3}	31	00000	00000	00000	566	0

¹ Optional instruction not implemented by the e600 core.

² Optional to the PowerPC architecture but implemented by the e600 core.

³ Supervisor-level instruction.

Table A-32. Vector Integer Arithmetic Instructions

Name	0	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
vaddcuw ¹	04		vD		vA		vB															384						0
vaddsbs ¹	04		vD		vA		vB															768						0
vaddshs ¹	04		vD		vA		vB															832						0
vaddsws ¹	04		vD		vA		vB															896						0
vaddubm ¹	04		vD		vA		vB															0						0
vaddubs ¹	04		vD		vA		vB															512						0
vadduhm ¹	04		vD		vA		vB															64						0
vadduhs ¹	04		vD		vA		vB															576						0
vadduwm ¹	04		vD		vA		vB															128						0
vadduws ¹	04		vD		vA		vB															640						0
vavgsb ¹	04		vD		vA		vB															1282						0
vavgsh ¹	04		vD		vA		vB															1346						0
vavgsw ¹	04		vD		vA		vB															1410						0
vavgub ¹	04		vD		vA		vB															1026						0
vavguh ¹	04		vD		vA		vB															1090						0
vavguw ¹	04		vD		vA		vB															1154						0
vmaxsb ¹	04		vD		vA		vB															258						
vmaxsh ¹	04		vD		vA		vB															322						
vmaxsw ¹	04		vD		vA		vB															386						
vmaxub ¹	04		vD		vA		vB															2						
vmaxuh ¹	04		vD		vA		vB															66						
vmaxuw ¹	04		vD		vA		vB															130						
vmhaddshs ¹	04		vD		vA		vB													vC						32		
vmhraddshs ¹	04		vD		vA		vB													vC						33		
vminsb ¹	04		vD		vA		vB															770						
vminsh ¹	04		vD		vA		vB															834						
vminsw ¹	04		vD		vA		vB															898						
vminub ¹	04		vD		vA		vB															514						
vminuh ¹	04		vD		vA		vB															578						
vminuw ¹	04		vD		vA		vB															642						
vmladduhm ¹	04		vD		vA		vB														vC					34		
vmsummbm ¹	04		vD		vA		vB														vC					37		

Table A-32. Vector Integer Arithmetic Instructions (continued)

Name	0	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	
vmsumshm ¹	04		vD		vA		vB		vC																				40
vmsumshs ¹	04		vD		vA		vB		vC																				41
vmsumubm ¹	04		vD		vA		vB		vC																				36
vmsumuhm ¹	04		vD		vA		vB		vC																				38
vmsumuhs ¹	04		vD		vA		vB		vC																				39
vmulesb ¹	04		vD		vA		vB																						776
vmulesh ¹	04		vD		vA		vB																						840
vmuleub ¹	04		vD		vA		vB																						520
vmuleuh ¹	04		vD		vA		vB																						584
vmulosb ¹	04		vD		vA		vB																						264
vmulosh ¹	04		vD		vA		vB																						328
vmuloub ¹	04		vD		vA		vB																						8
vmulouh ¹	04		vD		vA		vB																						72
vsubcuw ¹	04		vD		vA		vB																						1408
vsubsbs ¹	04		vD		vA		vB																						1792
vsubshs ¹	04		vD		vA		vB																						1856
vsubsws ¹	04		vD		vA		vB																						1920
vsububm ¹	04		vD		vA		vB																						1024
vsububs ¹	04		vD		vA		vB																						1536
vsubuhm ¹	04		vD		vA		vB																						1088
vsubuhs ¹	04		vD		vA		vB																						1600
vsubuwm ¹	04		vD		vA		vB																						1152
vsubuws ¹	04		vD		vA		vB																						1664
vsumsws ¹	04		vD		vA		vB																						1928
vsum2sws ¹	04		D		A		B																						1672
vsum4sbs ¹	04		D		A		B																						1800
vsum4shs ¹	04		D		A		B																						1608
vsum4ubs ¹	04		D		A		B																						1544

¹ AltiVec technology-specific instruction

Table A-33. Floating-Point Compare Instructions

Name	0	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
vcmpbfp	04			vD						vA									vB	Rc								966
vcmpqfp	04			vD						vA									vB	Rc								198
vcmpgfp	04			vD						vA									vB	Rc								454
vcmpgtfp	04			vD						vA									vB	Rc								710

Table A-34. Floating-Point Estimate Instructions

Name	0	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
vexptfp	04			vD															vB									394
vlogefp	04			vD															vB									458
vrefp	04			vD															vB									266
vrsqrtefp	04			vD															vB									330

Table A-35. Vector Load Instructions Supporting Alignment

Name	0	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	
lvsl	31			vD						A					B													6	0
lvsl	31			vD						A					B													38	0

Table A-36. Integer Store Instructions

Name	0	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	
stvebx	31			S						A					B													135	0
stvehx	31			S						A					B													167	0
stvewx	31			S						A					B													199	0
stvx	31			S						A					B													231	0
stvxl	31			S						A					B													487	0

Table A-37. Vector Pack Instructions

Name	0	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	
vpkpx	04			vD						vA					vB														782
vpkshss	04			vD						vA					vB														398
vpkshus	04			vD						vA					vB														270
vpkswss	04			vD						vA					vB														462
vpkswus	04			vD						vA					vB														334
vpkuhum	04			vD						vA					vB														14

Table A-37. Vector Pack Instructions (continued)

Name	0	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	
vpkuhus	04				vD					vA																			142
vpkuwum	04				vD					vA																			78
vpkuwus	04				vD					vA																			206

Table A-38. Vector Unpack Instructions

Name	0	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	
vmrghb	04				vD					vA																			12
vmrghh	04				vD					vA																			76
vmrghw	04				vD					vA																			140
vmrglb	04				vD					vA																			268
vmrglh	04				vD					vA																			332
vupkhp	04				D					0 0 0 0 0																			846
vupkhsb	04				D					0 0 0 0 0																			526
vupkhsh	04				D					0 0 0 0 0																			590
vupklp	04				D					0 0 0 0 0																			974
vupklsb	04				D					0 0 0 0 0																			654
vupklsh	04				D					0 0 0 0 0																			718

Table A-39. Vector Splat Instructions

Name	0	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	
vspltb	04				vD					UIMM																			524
vsplth	04				vD					UIMM																			588
vspltisb	04				vD					SIMM											0 0 0 0 0								780
vspltish	04				vD					SIMM											0 0 0 0 0								844
vspltisw	04				vD					SIMM											0 0 0 0 0								908
vspltw	04				vD					UIMM																			652

Table A-40. Vector Permute Instruction

Name	0	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	
vperm	04				vD					vA																			43

Table A-41. Vector Select Instruction

Name	0	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
vsel	04	vD			vA			vB			vC			42														

Table A-42. Vector Shift Instructions

Name	0	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
vsl	04	vD			vA			vB			452																	
vsldoi	04	vD			vA			vB			0	SH		44														
vslo	04	vD			vA			vB			1036																	
vsro	04	vD			vA			vB			1100																	

Table A-43. Move To/From Condition Register Instructions

Name	0	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
mfvscr	04	vD			00000			00000			1540						0											
mtvscr	04	00000			00000			vB			1604																	

Table A-44. User-Level Cache Instructions

Name	0	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
dss	31	A	00	STRM	00000			00000			822						0											
dssall	31	A	00	STRM	00000			00000			822						0											
dst	31	T	00	STRM	A			B			342						0											
dstst	31	T	00	STRM	A			B			374						0											
dststt	31	1	00	STRM	A			B			374						0											
dstt	31	1	00	STRM	A			B			342						0											

A.6 Instructions Sorted by Form

Table A-45 through Table A-58 list the e600 core instructions grouped by form.

Key:

Reserved bits

Table A-45. I-Form

OPCD	LI														AA	LK
------	----	--	--	--	--	--	--	--	--	--	--	--	--	--	----	----

Specific Instruction

Name	0	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
bx	18	LI														AA	LK											

Table A-46. B-Form

OPCD	BO	BI	BD	AA	LK
------	----	----	----	----	----

Specific Instruction

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

bcx

16	BO	BI	BD	AA	LK
----	----	----	----	----	----

Table A-47. SC-Form

OPCD	00000	00000	000000000000000000	1	0
------	-------	-------	--------------------	---	---

Specific Instruction

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

sc

17	00000	00000	000000000000000000	1	0
----	-------	-------	--------------------	---	---

Table A-48. D-Form

OPCD	D	A	d	
OPCD	D	A	SIMM	
OPCD	S	A	d	
OPCD	S	A	UIMM	
OPCD	crfD	0 L	A	SIMM
OPCD	crfD	0 L	A	UIMM
OPCD	TO	A	SIMM	

Table A-49. D-Form (continued)

Specific Instructions

Name	0	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
sth	45	S								A																		d
stmw¹	47	S								A																		d
stw	36	S								A																		d
stwu	37	S								A																		d
subfic	08	D								A																		SIMM
twi	03	TO								A																		SIMM
xori	26	S								A																		UIMM
xoris	27	S								A																		UIMM

¹ Load/store string/multiple instruction

Table A-49. X-Form

OPCD	D			A	B					XO		0
OPCD	D			A	NB					XO		0
OPCD	D			0 0 0 0 0	B					XO		0
OPCD	D			0 0 0 0 0	0 0 0 0 0					XO		0
OPCD	D		0	SR	0 0 0 0 0					XO		0
OPCD	S			A	B					XO		Rc
OPCD	S			A	B					XO		1
OPCD	S			A	B					XO		0
OPCD	S			A	NB					XO		0
OPCD	S			A	0 0 0 0 0					XO		Rc
OPCD	S			0 0 0 0 0	B					XO		0
OPCD	S			0 0 0 0 0	0 0 0 0 0					XO		0
OPCD	S		0	SR	0 0 0 0 0					XO		0
OPCD	S			A	SH					XO		Rc
OPCD	crfD		0	L	A	B				XO		0
OPCD	crfD		0 0		A	B				XO		0
OPCD	crfD		0 0	crfS	0 0	0 0 0 0 0				XO		0
OPCD	crfD		0 0	0 0 0 0 0	0 0 0 0 0					XO		0
OPCD	crfD		0 0	0 0 0 0 0	IMM	0				XO		Rc
OPCD	TO			A	B					XO		0
OPCD	D			0 0 0 0 0	B					XO		Rc

Table A-49. X-Form (continued)

OPCD	D			0 0 0 0 0	0 0 0 0 0	XO	Rc
OPCD	crbD			0 0 0 0 0	0 0 0 0 0	XO	Rc
OPCD	0 0 0 0 0			A	B	XO	0
OPCD	0 0 0 0 0			0 0 0 0 0	B	XO	0
OPCD	0 0 0 0 0			0 0 0 0 0	0 0 0 0 0	XO	0
OPCD	vD			vA	vB	XO	0
OPCD	vS			vA	vB	XO	0
OPCD	T	0 0	STRM	A	B	XO	0

Table A-50. X-Form

Specific Instructions

Name	0	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
andx	31	S			A			B			28			Rc														
andcx	31	S			A			B			60			Rc														
cmp	31	crfD	0	L	A			B			0			0														
cmpl	31	crfD	0	L	A			B			32			0														
cntlzwx	31	S			A			0 0 0 0 0			26			Rc														
dcba ¹	31	0 0 0 0 0			A			B			758			0														
dcbf	31	0 0 0 0 0			A			B			86			0														
dcbi ²	31	0 0 0 0 0			A			B			470			0														
dcbst	31	0 0 0 0 0			A			B			54			0														
dcbt	31	0 0 0 0 0			A			B			278			0														
dcbtst	31	0 0 0 0 0			A			B			246			0														
dcbz	31	0 0 0 0 0			A			B			1014			0														
dst	31	T	0 0	STRM	A			B			342			0														
dstt ³	31	1	0 0	STRM	A			B			342			0														
dstst ³	31	T	0 0	STRM	A			B			374			0														
dststt ³	31	1	0 0	STRM	A			B			374			0														
dss ³	31	A	0 0	STRM	0 0 0 0 0			0 0 0 0 0			822			0														
dssall ³	31	A	0 0	STRM	0 0 0 0 0			0 0 0 0 0			822			0														
eciwx ¹	31	D			A			B			310			0														
ecowx ¹	31	S			A			B			438			0														
eieio	31	0 0 0 0 0			0 0 0 0 0			0 0 0 0 0			854			0														
eqvx	31	S			A			B			284			Rc														

Table A-50. X-Form (continued)

Specific Instructions

Name	0	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	
extsbx	31				S						A			0	0	0	0	0											Rc
extshx	31				S						A			0	0	0	0	0											Rc
fabsx	63				D						0	0	0	0	0	0		B											Rc
fcmpo	63			crfD		0	0				A							B											0
fcmpu	63			crfD		0	0				A							B											0
fctiw_x	63				D						0	0	0	0	0	0		B											Rc
fctiwz_x	63				D						0	0	0	0	0	0		B											Rc
fmr_x	63				D						0	0	0	0	0	0		B											Rc
fnabs_x	63				D						0	0	0	0	0	0		B											Rc
fneg_x	63				D						0	0	0	0	0	0		B											Rc
frsp_x	63				D						0	0	0	0	0	0		B											Rc
icbi	31					0	0	0	0	0		A						B											0
lbz_{ux}	31				D						A							B											0
lbz_x	31				D						A							B											0
lfd_{ux}	31				D						A							B											0
lfd_x	31				D						A							B											0
lfs_{ux}	31				D						A							B											0
lfs_x	31				D						A							B											0
lhau_x	31				D						A							B											0
lhax	31				D						A							B											0
lhbr_x	31				D						A							B											0
lhzu_x	31				D						A							B											0
lhz_x	31				D						A							B											0
lsw i⁴	31				D						A							NB											0
lsw_x⁴	31				D						A							B											0
lveb_x³	31				vD						vA							vB											0
lveh_x³	31				vD						A							B											0
lvew_x³	31				vD						A							B											0
lvsl³	31				vD						A							B											0
lvsl³	31				vD						A							B											0

Table A-50. X-Form (continued)

Specific Instructions

Name	0	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	
stfsux	31					S					A																		0
stfsx	31					S					A																		0
sthbrx	31					S					A																		0
sthux	31					S					A																		0
sthx	31					S					A																		0
stswi ⁴	31					S					A							NB											0
stswx ⁴	31					S					A							B											0
stvebx ³	31					vS					A							B											0
stvehx ³	31					vS					A							B											0
stviewx ³	31					vS					A							B											0
stvx ³	31					vS					A							B											0
stvxl ³	31					vS					A							B											0
stwbrx ³	31					S					A							B											0
stwcx.	31					S					A							B											1
stwux	31					S					A							B											0
stwx	31					S					A							B											0
sync	31					0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0							0
tlbia ⁵	31					0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0							0
tlbie ²	31					0	0	0	0	0	0	0	0	0	0	0	0	B											0
tlbld ^{1,2}	31					0	0	0	0	0	0	0	0	0	0	0	0	B											0
tlbli ^{1,2}	31					0	0	0	0	0	0	0	0	0	0	0	0	B											0
tlbsync ²	31					0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0							0
tw	31					TO					A							B											0
xorx	31					S					A							B											Rc

¹ Optional to the PowerPC architecture but implemented by the e600 core.

² Supervisor-level instruction.

³ AltiVec technology-specific instruction.

⁴ Load/store string/multiple instruction.

⁵ Optional instruction not implemented by the e600 core.

Table A-50. XL-Form

OPCD	BO		BI		0 0 0 0 0	XO	LK
OPCD	crbD		crbA		crbB	XO	0
OPCD	crfD	0 0	crfS	0 0	0 0 0 0 0	XO	0
OPCD	0 0 0 0 0		0 0 0 0 0		0 0 0 0 0	XO	0

Specific Instructions

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

bcctrx	19	BO		BI		0 0 0 0 0	528	LK
bclrx	19	BO		BI		0 0 0 0 0	16	LK
crand	19	crbD		crbA		crbB	257	0
crandc	19	crbD		crbA		crbB	129	0
creqv	19	crbD		crbA		crbB	289	0
crnand	19	crbD		crbA		crbB	225	0
crnor	19	crbD		crbA		crbB	33	0
cror	19	crbD		crbA		crbB	449	0
crorc	19	crbD		crbA		crbB	417	0
crxor	19	crbD		crbA		crbB	193	0
isync	19	0 0 0 0 0		0 0 0 0 0		0 0 0 0 0	150	0
mcrf	19	crfD	0 0	crfS	0 0	0 0 0 0 0	0	0
rfi¹	19	0 0 0 0 0		0 0 0 0 0		0 0 0 0 0	50	0

¹ Supervisor-level instruction

Table A-51. XFX-Form

OPCD	D	spr				XO	0
OPCD	D	0	CRM			0	0
OPCD	S	spr				XO	0
OPCD	D	tbr				XO	0

Specific Instructions

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

mfspr¹	31	D	spr				339	0
mftb	31	D	tbr				371	0
mtrf	31	S	0	CRM			0	0
mtspr¹	31	D	spr				467	0

¹ Supervisor- and user-level instruction

Table A-52. XFL-Form

OPCD	0	FM	0	B	XO	Rc
------	---	----	---	---	----	----

Specific Instructions

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

mtfsfx	63	0	FM	0	B	711	Rc
---------------	----	---	----	---	---	-----	----

Table A-53. XO-Form

OPCD	D	A	B	OE	XO	Rc
OPCD	D	A	B	0	XO	Rc
OPCD	D	A	0 0 0 0 0	OE	XO	Rc

Specific Instructions

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

addx	31	D	A	B	OE	266	Rc
addcx	31	D	A	B	OE	10	Rc
addex	31	D	A	B	OE	138	Rc
addmex	31	D	A	0 0 0 0 0	OE	234	Rc
addzex	31	D	A	0 0 0 0 0	OE	202	Rc
divwx	31	D	A	B	OE	491	Rc
divwux	31	D	A	B	OE	459	Rc
mulhw_x	31	D	A	B	0	75	Rc
mulhwu_x	31	D	A	B	0	11	Rc
mullw_x	31	D	A	B	OE	235	Rc
neg_x	31	D	A	0 0 0 0 0	OE	104	Rc
subfx	31	D	A	B	OE	40	Rc
subfc_x	31	D	A	B	OE	8	Rc
subfex	31	D	A	B	OE	136	Rc
subfmex	31	D	A	0 0 0 0 0	OE	232	Rc
subfzex	31	D	A	0 0 0 0 0	OE	200	Rc

Table A-54. A-Form

OPCD	D	A	B	0 0 0 0 0	XO	Rc
OPCD	D	A	B	C	XO	Rc
OPCD	D	A	0 0 0 0 0	C	XO	Rc
OPCD	D	0 0 0 0 0	B	0 0 0 0 0	XO	Rc

Specific Instructions

Name	0	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
fadd_x	63	D	A	B	0 0 0 0 0																		21					Rc
fadds_x	59	D	A	B	0 0 0 0 0																			21				Rc
fdiv_x	63	D	A	B	0 0 0 0 0																			18				Rc
fdivs_x	59	D	A	B	0 0 0 0 0																			18				Rc
fmadd_x	63	D	A	B	C																			29				Rc
fmadds_x	59	D	A	B	C																			29				Rc
fmsub_x	63	D	A	B	C																			28				Rc
fmsubs_x	59	D	A	B	C																			28				Rc
fmul_x	63	D	A	0 0 0 0 0	C																			25				Rc
fmuls_x	59	D	A	0 0 0 0 0	C																			25				Rc
fnmadd_x	63	D	A	B	C																			31				Rc
fnmadds_x	59	D	A	B	C																			31				Rc
fnmsub_x	63	D	A	B	C																			30				Rc
fnmsubs_x	59	D	A	B	C																			30				Rc
fres_x ¹	59	D	0 0 0 0 0	B	0 0 0 0 0																			24				Rc
frsqrte_x ¹	63	D	0 0 0 0 0	B	0 0 0 0 0																			26				Rc
fsel_x ¹	63	D	A	B	C																			23				Rc
fsqrt_x ²	63	D	0 0 0 0 0	B	0 0 0 0 0																			22				Rc
fsqrts_x ²	59	D	0 0 0 0 0	B	0 0 0 0 0																			22				Rc
fsub_x	63	D	A	B	0 0 0 0 0																			20				Rc
fsubs_x	59	D	A	B	0 0 0 0 0																			20				Rc

¹ Optional to the PowerPC architecture but implemented by the e600 core.

² Optional instruction not implemented by the e600 core.

Table A-55. M-Form

OPCD	S	A	SH	MB	ME	Rc
OPCD	S	A	B	MB	ME	Rc

Specific Instructions

Name	0	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
rlwimix	20		S		A				SH											MB					ME			Rc
rlwinmx	21		S		A				SH											MB					ME			Rc
rlwnmx	23		S		A															B					ME			Rc

Table A-56. VA-Form

OPCD	vD	vA	vB		vC	XO
OPCD	vD	vA	vB	0	SH	XO

Specific Instructions

Name	0	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	
vmhaddshs ¹	04		vD		vA					vB											vC								32
vmhraddshs ¹	04		vD		vA					vB											vC								33
vmladduhm ¹	04		vD		vA					vB											vC								34
vmsumubm ¹	04		vD		vA					vB											vC								36
vmsummbm ¹	04		vD		vA					vB											vC								37
vmsumuhm ¹	04		vD		vA					vB											vC								38
vmsumuhs ¹	04		vD		vA					vB											vC								39
vmsumshm ¹	04		vD		vA					vB											vC								40
vmsumshs ¹	04		vD		vA					vB											vC								41
vsel ¹	04		vD		vA					vB											vC								42
vperm ¹	04		vD		vA					vB											vC								43
vsldoi ¹	04		vD		vA					vB											0				SH				44
vmaddfp ¹	04		vD		vA					vB											vC								46
vmnsbfp ¹	04		vD		vA					vB											vC								47

¹ AltiVec technology-specific instruction

Specific Instructions

Name	0	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	
vmaxsw ¹	04		vD		vA		vB																						386
vminub ¹	04		vD		vA		vB																						514
vminuh ¹	04		vD		vA		vB																						578
vminuw ¹	04		vD		vA		vB																						642
vminsb ¹	04		vD		vA		vB																						770
vminsh ¹	04		vD		vA		vB																						834
vminsw ¹	04		vD		vA		vB																						898
vavgub ¹	04		vD		vA		vB																						1026
vavguh ¹	04		vD		vA		vB																						1090
vavguw ¹	04		vD		vA		vB																						1154
vavgusb ¹	04		vD		vA		vB																						1282
vavgsh ¹	04		vD		vA		vB																						1346
vavgsw ¹	04		vD		vA		vB																						1410
vrlb ¹	04		vD		vA		vB																						4
vrlh ¹	04		vD		vA		vB																						68
vrlw ¹	04		vD		vA		vB																						132
vslb ¹	04		vD		vA		vB																						260
vslh ¹	04		vD		vA		vB																						324
vslw ¹	04		vD		vA		vB																						388
vsl ¹	04		vD		vA		vB																						452
vsrb ¹	04		vD		vA		vB																						516
vsrh ¹	04		vD		vA		vB																						580
vsrw ¹	04		vD		vA		vB																						644
vsr ¹	04		vD		vA		vB																						708
vsrab ¹	04		vD		vA		vB																						772
vsrah ¹	04		vD		vA		vB																						836
vsraw ¹	04		vD		vA		vB																						900
vand ¹	04		vD		vA		vB																						1028
vandc ¹	04		vD		vA		vB																						1092
vor ¹	04		vD		vA		vB																						1156
vnor ¹	04		vD		vA		vB																						1284
mfvscr ¹	04		vD						0	0	0	0	0	0	0	0	0	0										1540	0

Specific Instructions

Name	0	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
mtvscr ¹	04		0	0	0	0	0	0	0	0	0	0	vB									1604						0
vmuloub ¹	04		vD				vA						vB										8					
vmulouh ¹	04		vD				vA						vB										72					
vmulosb ¹	04		vD				vA						vB										264					
vmulosh ¹	04		vD				vA						vB										328					
vmuleub ¹	04		vD				vA						vB										520					
vmuleuh ¹	04		vD				vA						vB										584					
vmulesb ¹	04		vD				vA						vB										776					
vmulesh ¹	04		vD				vA						vB										840					
vsum4ubs ¹	04		vD				vA						vB										1544					
vsum4sbs ¹	04		vD				vA						vB										1800					
vsum4shs ¹	04		vD				vA						vB										1608					
vsum2sws ¹	04		vD				vA						vB										1672					
vsumsws ¹	04		vD				vA						vB										1928					
vaddfp ¹	04		vD				vA						vB										10					
vsubfp ¹	04		vD				vA						vB										74					
vrefp ¹	04		vD					0	0	0	0	0	vB										266					
vrsqrtefp ¹	04		vD					0	0	0	0	0	vB										330					
vexptefp ¹	04		vD					0	0	0	0	0	vB										394					
vlogefp ¹	04		vD					0	0	0	0	0	vB										458					
vrfin ¹	04		vD					0	0	0	0	0	vB										522					
vrfiz ¹	04		vD					0	0	0	0	0	vB										586					
vrfig ¹	04		vD					0	0	0	0	0	vB										650					
vrfig ¹	04		vD					0	0	0	0	0	vB										714					
vcfux ¹	04		vD										UIMM										778					
vcfsx ¹	04		vD										UIMM										842					
vctuxs ¹	04		vD										UIMM										906					
vctsxs ¹	04		vD										UIMM										970					
vmaxfp ¹	04		vD				vA						vB										1034					
vminf ¹	04		vD				vA						vB										1098					
vmrghb ¹	04		vD				vA						vB										12					
vmrghh ¹	04		vD				vA						vB										76					

Specific Instructions

Name	0	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
vmrghw ¹		04			vD				vA				vB															140
vmrglb ¹		04			vD				vA				vB															268
vmrglh ¹		04			vD				vA				vB															332
vmrglw ¹		04			vD				vA				vB															396
vspltb ¹		04			vD				UIMM				vB															524
vsplth ¹		04			vD				UIMM				vB															588
vspltw ¹		04			vD				UIMM				vB															652
vspltisb ¹		04			vD				SIMM				0	0	0	0	0											780
vspltish ¹		04			vD				SIMM				0	0	0	0	0											844
vspltisw ¹		04			vD				SIMM				0	0	0	0	0											908
vslo ¹		04			vD				vA				vB															1036
vsro ¹		04			vD				vA				vB															1100
vpkuhum ¹		04			vD				vA				vB															14
vpkuwum ¹		04			vD				vA				vB															78
vpkuhus ¹		04			vD				vA				vB															142
vpkuwus ¹		04			vD				vA				vB															206
vpkshus ¹		04			vD				vA				vB															270
vpkswus ¹		04			vD				vA				vB															334
vpkshs ¹		04			vD				vA				vB															398
vpkswss ¹		04			vD				vA				vB															462
vpkswus ¹		04			vD				vA				vB															334
vupkhsb ¹		04			vD				0	0	0	0	0					vB										526
vupkhsh ¹		04			vD				0	0	0	0	0					vB										590
vupklsb ¹		04			vD				0	0	0	0	0					vB										654
vupklsh ¹		04			vD				0	0	0	0	0					vB										718
vpkpx ¹		04			vD				vA				vB						12							782		
vupkhp ¹		04			vD				0	0	0	0	0					vB										846
vupklp ¹		04			vD				0	0	0	0	0					vB										974
vxor ¹		04			vD				vA				vB															1220

¹ AltiVec technology-specific instruction

Table A-58. VXR-Form

OPCD	vD	vA	vB	Rc	XO
------	----	----	----	----	----

Specific Instructions

Name	05	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
vcmpbfp_x ¹	04		vD			vA					vB			Rc													966
vcmpeqfp_x ¹	04		vD			vA					vB			Rc													198
vcmpequb_x ¹	04		vD			vA					vB			Rc													6
vcmpequh_x ¹	04		vD			vA					vB			Rc													70
vcmpequw_x ¹	04		vD			vA					vB			Rc													134
vcmpgef_x ¹	04		vD			vA					vB			Rc													454
vcmpgtf_x ¹	04		vD			vA					vB			Rc													710
vcmpgtsb_x ¹	04		vD			vA					vB			Rc													774
vcmpgtsh_x ¹	04		vD			vA					vB			Rc													838
vcmpgtsw_x ¹	04		vD			vA					vB			Rc													902
vcmpgtub_x ¹	04		vD			vA					vB			Rc													518
vcmpgtuh_x ¹	04		vD			vA					vB			Rc													582
vcmpgtuw_x ¹	04		vD			vA					vB			Rc													646

¹ AltiVec technology-specific instruction

A.7 Instruction Set Legend

Table A-59 provides general information on the PowerPC instruction set (such as the architectural level, privilege level, and form).

Table A-59. PowerPC Instruction Set Legend

Name	UISA	VEA	OEA	Supervisor Level	Optional	Form
add_x	√					XO
addc_x	√					XO
addex	√					XO
addi	√					D
addic	√					D
addic.	√					D
addis	√					D
addmex	√					XO
addzex	√					XO
and_x	√					X

Table A-59. PowerPC Instruction Set Legend (continued)

Name	UISA	VEA	OEA	Supervisor Level	Optional	Form
andcx	√					X
andi.	√					D
andis.	√					D
bx	√					I
bcx	√					B
bcctrx	√					XL
bclrx	√					XL
cmp	√					X
cmpi	√					D
cmpl	√					X
cmpli	√					D
cntlzwx	√					X
crand	√					XL
crandc	√					XL
creqv	√					XL
crnand	√					XL
crnor	√					XL
cror	√					XL
crorc	√					XL
crxor	√					XL
dcba		√			√	X
dcbf		√				X
dcbi			√	√		X
dcbst		√				X
dcbt		√				X
dcbtst		√				X
dcbz		√				X
divwx	√					XO
divwux	√					XO
eciwx		√			√	X
ecowx		√			√	X
eieio		√				X
eqvx	√					X

Table A-59. PowerPC Instruction Set Legend (continued)

Name	UISA	VEA	OEA	Supervisor Level	Optional	Form
extsbx	√					X
extshx	√					X
fabsx	√					X
faddx	√					A
faddsx	√					A
fcmpo	√					X
fcmpu	√					X
fctiw _x	√					X
fctiwz _x	√					X
fdiv _x	√					A
fdivs _x	√					A
fmadd _x	√					A
fmaddsx	√					A
fmr _x	√					X
fmsub _x	√					A
fmsubs _x	√					A
fmul _x	√					A
fmul _{sx}	√					A
fnabs _x	√					X
fneg _x	√					X
fnmadd _x	√					A
fnmaddsx	√					A
fnmsub _x	√					A
fnmsubs _x	√					A
fres _x	√				√	A
frsp _x	√					X
frsqrte _x	√				√	A
fsel _x	√				√	A
fsqrt _x	√				√	A
fsqrts _x	√				√	A
fsub _x	√					A
fsubs _x	√					A
icbi		√				X

Table A-59. PowerPC Instruction Set Legend (continued)

Name	UISA	VEA	OEA	Supervisor Level	Optional	Form
isync		√				XL
lbz	√					D
lbzu	√					D
lbzux	√					X
lbzx	√					X
lfd	√					D
lfdv	√					D
lfdvux	√					X
lfdx	√					X
lfs	√					D
lfsu	√					D
lfsux	√					X
lfsx	√					X
lha	√					D
lhau	√					D
lhauX	√					X
lhax	√					X
lhbrx	√					X
lhz	√					D
lhzu	√					D
lhzux	√					X
lhzx	√					X
lmw ²	√					D
lswi ²	√					X
lswx ²	√					X
lwarx	√					X
lwbrx	√					X
lwz	√					D
lwzu	√					D
lwzux	√					X
lwzx	√					X
mcrf	√					XL
mcrfs	√					X

Table A-59. PowerPC Instruction Set Legend (continued)

Name	UISA	VEA	OEA	Supervisor Level	Optional	Form
mcrxr	√					X
mfcrr	√					X
mffsx	√					X
mfmsr			√	√		X
mfspr ¹	√		√	√		AFX
mfsr			√	√		X
mfsrin			√	√		X
mftb		√				AFX
mtcrf	√					AFX
mtfsb0x	√					X
mtfsb1x	√					X
mtfsfx	√					AXL
mtfsfix	√					X
mtmsr			√	√		X
mtspr ¹	√		√	√		AFX
mtsr			√	√		X
mtsrin			√	√		X
mulhw _x	√					XO
mulhw _x	√					XO
mulli	√					D
mullw _x	√					XO
nand _x	√					X
neg _x	√					XO
nor _x	√					X
or _x	√					X
orc _x	√					X
ori	√					D
oris	√					D
rfi			√	√		XL
rlwim _x	√					M
rlwinm _x	√					M
rlwnm _x	√					M

Table A-59. PowerPC Instruction Set Legend (continued)

Name	UISA	VEA	OEA	Supervisor Level	Optional	Form
sc	√		√			SC
slwx	√					X
srawx	√					X
srawix	√					X
srwx	√					X
stb	√					D
stbu	√					D
stbux	√					X
stbx	√					X
stfd	√					D
stfdu	√					D
stfdux	√					X
stfdx	√					X
stfiwx	√					X
stfs	√					D
stfsu	√					D
stfsux	√					X
stfsx	√					X
sth	√					D
sthbrx	√					X
sthu	√					D
sthux	√					X
sthx	√					X
stmw ²	√					D
stswi ²	√					X
stswx ²	√					X
stw	√					D
stwbrx	√					X
stwcx.	√					X
stwu	√					D
stwux	√					X
stwx	√					X
subfx	√					XO

Table A-59. PowerPC Instruction Set Legend (continued)

Name	UISA	VEA	OEA	Supervisor Level	Optional	Form
subfcx	√					XO
subfex	√					XO
subfic	√					D
subfmex	√					XO
subfzex	√					XO
sync	√					X
tlbia			√	√	√	X
tlbie			√	√	√	X
tlbsync			√	√		X
tw	√					X
twi	√					D
xorx	√					X
xori	√					D
xoris	√					D

Notes:

- ¹ Supervisor- and user-level instruction.
- ² Load/store string or multiple instruction.
- ³ Optional instruction provided to support temporary 64-bit bridge.
- ⁴ Defined for the 32-bit architecture and by the temporary 64-bit bridge.



Appendix B Instructions Not Implemented

This appendix provides a list of the 32-bit instructions that are not implemented in the e600 core. Note that an attempt to execute instructions that are not implemented on the e600 core generates an illegal instruction program interrupt.

[Table B-1](#) provides the 32-bit PowerPC instructions that are optional to the PowerPC architecture and not implemented by the e600 core.

Table B-1. 32-Bit Instructions Not Implemented by the e600 Core

Mnemonic	Instruction
fsqrtx	Floating Square Root (Double-Precision)
fsqrtsx	Floating Square Root Single
tbia	Translation Lookaside Buffer Invalidate All

Appendix C Special-Purpose Registers

Table C-1. PowerPC SPR Encodings Ordered by Decimal Value

Register Name	SPR ¹			Access	mfspr/mtspr
	Decimal	spr[5–9]	spr[0–4]		
XER	1	00000	00001	User (UISA)	Both
LR	8	00000	01000	User (UISA)	Both
CTR	9	00000	01001	User (UISA)	Both
DSISR	18	00000	10010	Supervisor (OEA)	Both
DAR	19	00000	10011	Supervisor (OEA)	Both
DEC	22	00000	10110	Supervisor (OEA)	Both
SDR1	25	00000	11001	Supervisor (OEA)	Both
SRR0	26	00000	11010	Supervisor (OEA)	Both
SRR1	27	00000	11011	Supervisor (OEA)	Both
VRSAVE ²	256	01000	00000	User (AltiVec/UISA)	Both
TBL ³	268	01000	01100	User (VEA)	mfspr, mftb
TBU ³	269	01000	01101	User (VEA)	mfspr, mftb
SPRG0	272	01000	10000	Supervisor (OEA)	Both
SPRG1	273	01000	10001	Supervisor (OEA)	Both
SPRG2	274	01000	10010	Supervisor (OEA)	Both
SPRG3	275	01000	10011	Supervisor (OEA)	Both
SPRG4 ⁴	276	01000	10100	Supervisor (OEA)	Both
SPRG5 ⁴	277	01000	10101	Supervisor (OEA)	Both
SPRG6 ⁴	278	01000	100110	Supervisor (OEA)	Both
SPRG7 ⁴	279	01000	10111	Supervisor (OEA)	Both
EAR ⁵	282	01000	11010	Supervisor (OEA)	Both
TBL ³	284	01000	11100	Supervisor (OEA)	mtspr
TBU ³	285	01000	11101	Supervisor (OEA)	mtspr
SVR	286	01000	11110	Supervisor (OEA)	mfspr
PVR	287	01000	11111	Supervisor (OEA)	mfspr

Table C-1. PowerPC SPR Encodings Ordered by Decimal Value (continued)

Register Name	SPR ¹			Access	mfspr/mtspr
	Decimal	spr[5–9]	spr[0–4]		
IBAT0U	528	10000	10000	Supervisor (OEA)	Both
IBAT0L	529	10000	10001	Supervisor (OEA)	Both
IBAT1U	530	10000	10010	Supervisor (OEA)	Both
IBAT1L	531	10000	10011	Supervisor (OEA)	Both
IBAT2U	532	10000	10100	Supervisor (OEA)	Both
IBAT2L	533	10000	10101	Supervisor (OEA)	Both
IBAT3U	534	10000	10110	Supervisor (OEA)	Both
IBAT3L	535	10000	10111	Supervisor (OEA)	Both
DBAT0U	536	10000	11000	Supervisor (OEA)	Both
DBAT0L	537	10000	11001	Supervisor (OEA)	Both
DBAT1U	538	10000	11010	Supervisor (OEA)	Both
DBAT1L	539	10000	11011	Supervisor (OEA)	Both
DBAT2U	540	10000	11100	Supervisor (OEA)	Both
DBAT2L	541	10000	11101	Supervisor (OEA)	Both
DBAT3U	542	10000	11110	Supervisor (OEA)	Both
DBAT3L	543	10000	11111	Supervisor (OEA)	Both
IBAT4U ⁴	560	10001	10000	Supervisor (OEA)	Both
IBAT4L ⁴	561	10001	10001	Supervisor (OEA)	Both
IBAT5U ⁴	562	10001	10010	Supervisor (OEA)	Both
IBAT5L ⁴	563	10001	10011	Supervisor (OEA)	Both
IBAT6U ⁴	564	10001	10100	Supervisor (OEA)	Both
IBAT6L ⁴	565	10001	10101	Supervisor (OEA)	Both
IBAT7U ⁴	566	10001	10110	Supervisor (OEA)	Both
IBAT7L ⁴	567	10001	10111	Supervisor (OEA)	Both
DBAT4U ⁴	568	10001	11000	Supervisor (OEA)	Both
DBAT4L ⁴	569	10001	11001	Supervisor (OEA)	Both
DBAT5U ⁴	570	10001	11010	Supervisor (OEA)	Both
DBAT5L ⁴	571	10001	11011	Supervisor (OEA)	Both
DBAT6U ⁴	572	10001	11100	Supervisor (OEA)	Both
DBAT6L ⁴	573	10001	11101	Supervisor (OEA)	Both
DBAT7U ⁴	574	10001	11110	Supervisor (OEA)	Both

Table C-1. PowerPC SPR Encodings Ordered by Decimal Value (continued)

Register Name	SPR ¹			Access	mfspr/mtspr
	Decimal	spr[5–9]	spr[0–4]		
DBAT7L ⁴	575	10001	11111	Supervisor (OEA)	Both
UMMCR2 ⁴	928	11101	00000	User (UISA)	mfspr
UPMC5 ⁴	929	11101	00001	User (UISA)	mfspr
UPMC6 ⁴	930	11101	00010	User (UISA)	mfspr
UMMCR0 ⁴	936	11101	01000	User (UISA)	mfspr
UPMC1 ⁴	937	11101	01001	User (UISA)	mfspr
UPMC2 ⁴	938	11101	01010	User (UISA)	mfspr
USIAR ⁴	939	11101	01011	User (UISA)	mfspr
UMMCR1 ⁴	940	11101	01100	User (UISA)	mfspr
UPMC3 ⁴	941	11101	01101	User (UISA)	mfspr
UPMC4 ⁴	942	11101	01110	User (UISA)	mfspr
MMCR2 ⁴	944	11101	10000	Supervisor (OEA)	Both
PMC5 ⁵	945	11101	10001	Supervisor (OEA)	Both
PMC6 ⁵	946	11101	10010	Supervisor (OEA)	Both
BAMR ⁴	951	11101	10111	Supervisor (OEA)	Both
MMCR0 ⁵	952	11101	11000	Supervisor (OEA)	Both
PMC1 ⁵	953	11101	11001	Supervisor (OEA)	Both
PMC2 ⁵	954	11101	11010	Supervisor (OEA)	Both
SIAR ⁵	955	11101	11011	Supervisor (OEA)	Both
MMCR1 ⁵	956	11101	11100	Supervisor (OEA)	Both
PMC3 ⁵	957	11101	11101	Supervisor (OEA)	Both
PMC4 ⁵	958	11101	11110	Supervisor (OEA)	Both
TLBMISS ³	980	11110	10100	Supervisor (OEA)	Both
PTEHI ⁴	981	11110	10101	Supervisor (OEA)	Both
PTELO ⁴	982	11110	10110	Supervisor (OEA)	Both
L2ERRINJHI	985	11110	11001	Supervisor (OEA)	Both
L2ERRINJLO	986	11110	11010	Supervisor (OEA)	Both
L2ERRINJCTL	987	11110	11011	Supervisor (OEA)	Both
L2CAPTDATAHI	988	11110	11100	Supervisor (OEA)	mfspr
L2CAPTDATALO	989	11110	11101	Supervisor (OEA)	mfspr
L2CAPTECC	990	11110	11110	Supervisor (OEA)	mfspr

Table C-1. PowerPC SPR Encodings Ordered by Decimal Value (continued)

Register Name	SPR ¹			Access	mfspr/mtspr
	Decimal	spr[5–9]	spr[0–4]		
L2ERRDET	991	11110	11111	Supervisor (OEA)	Special ⁶
L2ERRDIS	992	11111	00000	Supervisor (OEA)	Both
L2ERRINTEN	993	11111	00001	Supervisor (OEA)	Both
L2ERRATTR	994	11111	00010	Supervisor (OEA)	Both
L2ERRADDR	995	11111	00011	Supervisor (OEA)	mfspr
L2ERREADDR	996	11111	00100	Supervisor (OEA)	mfspr
L2ERRCTL	997	11111	00101	Supervisor (OEA)	Both
HID0 ⁴	1008	11111	10000	Supervisor (OEA)	Both
HID1	1009	11111	10001	Supervisor (OEA)	Both
IABR ⁴	1010	11111	10010	Supervisor (OEA)	Both
ICTRL ⁴	1011	11111	10011	Supervisor (OEA)	Both
DABR ⁵	1013	11111	10101	Supervisor (OEA)	Both
MSSCR0 ⁴	1014	11111	10110	Supervisor (OEA)	Both
MSSSR0	1015	11111	10111	Supervisor (OEA)	Both
LDSTCR ⁴	1016	11111	11000	Supervisor (OEA)	Both
L2CR	1017	11111	11001	Supervisor (OEA)	Both
ICTC ⁴	1019	11111	11011	Supervisor (OEA)	Both
PIR ⁵	1023	11111	11111	Supervisor (OEA)	Both

¹ Note that the order of the two 5-bit halves of the SPR number is reversed compared with actual instruction coding. For **mtspr** and **mfspr** instructions, the SPR number coded in assembly language does not appear directly as a 10-bit binary number in the instruction. The number coded is split into two 5-bit halves that are reversed in the instruction, with the high-order 5 bits appearing in bits 16–20 of the instruction and the low-order 5 bits in bits 11–15.

² Register defined by the AltiVec technology.

³ The TB registers are referred to as TBRs rather than SPRs and can be written to using the **mtspr** instruction in supervisor mode and the TBR numbers here. The TB registers can be read in user mode using either the **mftb** or **mtspr** instruction and specifying TBR 268 for TBL and TBR 269 for TBU.

⁴ e600-specific only, register may not be supported on other processors that implement the PowerPC architecture.

⁵ Register defined as optional in the PowerPC architecture.

⁶ Most bits are bit reset/write 1 clear. A write of 0 to a bit does not change it. A write of 1 to a bit clears it. Reads act normally.

Table C-2. PowerPC SPR Encodings Ordered by Register Name

Register Name	SPR ¹			Access	mfspr/mtspr
	Decimal	spr[5–9]	spr[0–4]		
BAMR ²	951	11101	10111	Supervisor (OEA)	Both
CTR	9	00000	01001	User (UISA)	Both
DABR ²	1013	11111	10101	Supervisor (OEA)	Both
DAR	19	00000	10011	Supervisor (OEA)	Both
DBAT0L	537	10000	11001	Supervisor (OEA)	Both
DBAT0U	536	10000	11000	Supervisor (OEA)	Both
DBAT1L	539	10000	11011	Supervisor (OEA)	Both
DBAT1U	538	10000	11010	Supervisor (OEA)	Both
DBAT2L	541	10000	11101	Supervisor (OEA)	Both
DBAT2U	540	10000	11100	Supervisor (OEA)	Both
DBAT3L	543	10000	11111	Supervisor (OEA)	Both
DBAT3U	542	10000	11110	Supervisor (OEA)	Both
DBAT4L ²	569	10001	11001	Supervisor (OEA)	Both
DBAT4U ⁴	568	10001	11000	Supervisor (OEA)	Both
DBAT5L ⁴	571	10001	11011	Supervisor (OEA)	Both
DBAT5U ⁴	570	10001	11010	Supervisor (OEA)	Both
DBAT6L ⁴	573	10001	11101	Supervisor (OEA)	Both
DBAT6U ⁴	572	10001	11100	Supervisor (OEA)	Both
DBAT7L ⁴	575	10001	11111	Supervisor (OEA)	Both
DBAT7U ⁴	574	10001	11110	Supervisor (OEA)	Both
DEC	22	00000	10110	Supervisor (OEA)	Both
DSISR	18	00000	10010	Supervisor (OEA)	Both
EAR ²	282	01000	11010	Supervisor (OEA)	Both
HID0 ²	1008	11111	10000	Supervisor (OEA)	Both
HID1	1009	11111	10001	Supervisor (OEA)	Both
IABR ²	1010	11111	10010	Supervisor (OEA)	Both
IBAT0L	529	10000	10001	Supervisor (OEA)	Both
IBAT0U	528	10000	10000	Supervisor (OEA)	Both
IBAT1L	531	10000	10011	Supervisor (OEA)	Both
IBAT1U	530	10000	10010	Supervisor (OEA)	Both
IBAT2L	533	10000	10101	Supervisor (OEA)	Both

Table C-2. PowerPC SPR Encodings Ordered by Register Name (continued)

Register Name	SPR ¹			Access	mfspr/mtspr
	Decimal	spr[5–9]	spr[0–4]		
IBAT2U	532	10000	10100	Supervisor (OEA)	Both
IBAT3L	535	10000	10111	Supervisor (OEA)	Both
IBAT3U	534	10000	10110	Supervisor (OEA)	Both
IBAT4L ⁴	561	10001	10001	Supervisor (OEA)	Both
IBAT4U ⁴	560	10001	10000	Supervisor (OEA)	Both
IBAT5L ⁴	563	10001	10011	Supervisor (OEA)	Both
IBAT5U ⁴	562	10001	10010	Supervisor (OEA)	Both
IBAT6L ⁴	565	10001	10101	Supervisor (OEA)	Both
IBAT6U ⁴	564	10001	10100	Supervisor (OEA)	Both
IBAT7L ⁴	567	10001	10111	Supervisor (OEA)	Both
IBAT7U ⁴	566	10001	10110	Supervisor (OEA)	Both
ICTC ²	1019	11111	11011	Supervisor (OEA)	Both
ICTRL ²	1011	11111	10011	Supervisor (OEA)	Both
L2CAPTECC ²	990	11110	11110	Supervisor (OEA)	mfspr
L2CAPTDATAHI ²	988	11110	11100	Supervisor (OEA)	mfspr
L2CAPTDATALO ²	989	11110	11101	Supervisor (OEA)	mfspr
L2CR	1017	11111	11001	Supervisor (OEA)	Both
L2ERRADDR ²	995	11111	00011	Supervisor (OEA)	mfspr
L2ERRATTR ²	994	11111	00010	Supervisor (OEA)	Both
L2ERRCTL ²	997	11111	00101	Supervisor (OEA)	Both
L2ERRDET ²	991	11110	11111	Supervisor (OEA)	Special ³
L2ERRDIS ²	992	11111	00000	Supervisor (OEA)	Both
L2ERREADDR ²	996	11111	00100	Supervisor (OEA)	mfspr
L2ERRINJCTL ²	987	11110	11011	Supervisor (OEA)	Both
L2ERRINJHI ²	985	11110	11001	Supervisor (OEA)	Both
L2ERRINJLO ²	986	11110	11010	Supervisor (OEA)	Both
L2ERRINTEN ²	993	11111	00001	Supervisor (OEA)	Both
LDSTCR ²	1016	11111	11000	Supervisor (OEA)	Both
LR	8	00000	01000	User (UISA)	Both
MMCR0 ²	952	11101	11000	Supervisor (OEA)	Both
MMCR1 ²	956	11101	11100	Supervisor (OEA)	Both

Table C-2. PowerPC SPR Encodings Ordered by Register Name (continued)

Register Name	SPR ¹			Access	mfspr/mtspr
	Decimal	spr[5–9]	spr[0–4]		
MMCR2 ²	944	11101	10000	Supervisor (OEA)	Both
MSSCR0 ²	1014	11111	10110	Supervisor (OEA)	Both
MSSSR0	1015	11111	10111	Supervisor (OEA)	Both
PIR ²	1023	11111	11111	Supervisor (OEA)	Both
PMC1 ²	953	11101	11001	Supervisor (OEA)	Both
PMC2 ²	954	11101	11010	Supervisor (OEA)	Both
PMC3 ²	957	11101	11101	Supervisor (OEA)	Both
PMC4 ²	958	11101	11110	Supervisor (OEA)	Both
PMC5 ²	945	11101	10001	Supervisor (OEA)	Both
PMC6 ²	946	11101	10010	Supervisor (OEA)	Both
PTEHI ²	981	11110	10101	Supervisor (OEA)	Both
PTELO ²	982	11110	10110	Supervisor (OEA)	Both
PVR	287	01000	11111	Supervisor (OEA)	mfspr
SDR1	25	00000	11001	Supervisor (OEA)	Both
SIAR ²	955	11101	11011	Supervisor (OEA)	Both
SPRG0	272	01000	10000	Supervisor (OEA)	Both
SPRG1	273	01000	10001	Supervisor (OEA)	Both
SPRG2	274	01000	10010	Supervisor (OEA)	Both
SPRG3	275	01000	10011	Supervisor (OEA)	Both
SPRG4 ⁴	276	01000	10100	Supervisor (OEA)	Both
SPRG5 ⁴	277	01000	10101	Supervisor (OEA)	Both
SPRG6 ⁴	278	01000	100110	Supervisor (OEA)	Both
SPRG7 ⁴	279	01000	10111	Supervisor (OEA)	Both
SRR0	26	00000	11010	Supervisor (OEA)	Both
SRR1	27	00000	11011	Supervisor (OEA)	Both
SVR ²	286	01000	11110	Supervisor (OEA)	mfspr
TBL ⁴	284	01000	11100	Supervisor (OEA)	mtspr
TBL ²	268	01000	01100	User (VEA)	mfspr, mftb
TBU ²	285	01000	11101	Supervisor (OEA)	mtspr
TBU ²	269	01000	01101	User (VEA)	mfspr, mftb
TLBMISS ²	980	11110	10100	Supervisor (OEA)	Both

Table C-2. PowerPC SPR Encodings Ordered by Register Name (continued)

Register Name	SPR ¹			Access	mfspr/mtspr
	Decimal	spr[5–9]	spr[0–4]		
UMMCR0 ²	936	11101	01000	User (UISA)	mfspr
UMMCR1 ²	940	11101	01100	User (UISA)	mfspr
UMMCR2 ²	928	11101	00000	User (UISA)	mfspr
UPMC1 ²	937	11101	01001	User (UISA)	mfspr
UPMC2 ²	938	11101	01010	User (UISA)	mfspr
UPMC3 ²	941	11101	01101	User (UISA)	mfspr
UPMC4 ²	942	11101	01110	User (UISA)	mfspr
UPMC5 ²	929	11101	00001	User (UISA)	mfspr
UPMC6 ²	930	11101	00010	User (UISA)	mfspr
USIAR ²	939	11101	01011	User (UISA)	mfspr
VRSAVE ⁵	256	01000	00000	User (AltiVec/UISA)	Both
XER	1	00000	00001	User (UISA)	Both

¹ Note that the order of the two 5-bit halves of the SPR number is reversed compared with actual instruction coding. For **mtspr** and **mfspr** instructions, the SPR number coded in assembly language does not appear directly as a 10-bit binary number in the instruction. The number coded is split into two 5-bit halves that are reversed in the instruction, with the high-order 5 bits appearing in bits 16–20 of the instruction and the low-order 5 bits in bits 11–15.

² Register defined as optional in the PowerPC architecture.

³ Most bits are bit reset/write 1 clear. A write of 0 to a bit does not change it. A write of 1 to a bit clears it. Reads act normally.

⁴ The TB registers are referred to as TBRs rather than SPRs and can be written to using the **mtspr** instruction in supervisor mode and the TBR numbers here. The TB registers can be read in user mode using either the **mftb** or **mtspr** instruction and specifying TBR 268 for TBL and TBR 269 for TBU.

⁵ Register defined by the AltiVec technology.

Appendix D

Revision History

This appendix provides a list of the major differences between revisions of the *e600 PowerPC™ Core Reference Manual*. This is the initial version of the manual so there are currently no differences.

Glossary

The glossary contains an alphabetical list of terms, phrases, and abbreviations used in this reference manual.

A

Architecture. A detailed specification of requirements for a processor or computer system. It does not specify details of how the processor or computer system must be implemented; instead it provides a template for a family of compatible *implementations*.

Atomic access. A bus access that attempts to be part of a read-write operation to the same address uninterrupted by any other access to that address (the term refers to the fact that the transactions are indivisible). The PowerPC architecture implements atomic accesses through the **lwarx/stwcx** instruction pair.

Autobaud. The process of determining a serial data rate by timing the width of a single bit.

B

Beat. A single state on the bus interface that may extend across multiple bus cycles. A transaction can be composed of multiple address or data *beats*.

Big endian. A byte-ordering method in memory where the address *n* of a word corresponds to the *most significant byte*. In an addressed memory word, the bytes are ordered (left to right) 0, 1, 2, 3, with 0 being the *most significant byte*. See *Little endian*.

Boundedly undefined. A characteristic of certain operation results that are not rigidly prescribed by the PowerPC architecture. Boundedly-undefined results for a given operation may vary among implementations and between execution attempts in the same implementation.

Although the architecture does not prescribe the exact behavior for when results are allowed to be boundedly undefined, the results of executing instructions in contexts where results are allowed to be boundedly undefined are constrained to ones that could have been achieved by executing an arbitrary sequence of defined instructions, in valid form, starting in the state the machine was in before attempting to execute the given instruction.

Breakpoint. A programmable event that forces the core to take a breakpoint interrupt.

Burst. A multiple-beat data transfer whose total size is typically equal to a cache block.

Bus clock. Clock that causes the bus state transitions.

Bus master. The owner of the address or data bus; the device that initiates or requests the transaction.

C

Cache. High-speed memory containing recently accessed data or instructions (subset of main memory).

Cache block. A small region of contiguous memory that is copied from memory into a *cache*. The size of a cache block may vary among processors; the maximum block size is one *page*. In PowerPC processors, *cache coherency* is maintained on a cache-block basis. Note that the term ‘cache block’ is often used interchangeably with ‘cache line.’

Cache coherency. An attribute wherein an accurate and common view of memory is provided to all devices that share the same memory system. Caches are coherent if a processor performing a read from its cache is supplied with data corresponding to the most recent value written to memory or to another processor’s cache.

Cache flush. An operation that removes from a cache any data from a specified address range. This operation ensures that any modified data within the specified address range is written back to main memory. This operation is generated typically by a Data Cache Block Flush (**dcbf**) instruction.

Caching-inhibited. A memory update policy in which the *cache* is bypassed and the load or store is performed to or from main memory.

Cast out. A *cache block* that must be written to memory when a cache miss causes a cache block to be replaced.

Changed bit. One of two *page history bits* found in each *page table entry* (PTE). The processor sets the changed bit if any store is performed into the *page*. See also *Page access history bits* and *Referenced*.

Clean. An operation that causes a cache block to be written to memory, if modified, and then left in a valid, unmodified state in the cache.

Clear. To cause a bit or bit field to register a value of zero. See also *Set*.

Completer. In PCI-X, a completer is the device addressed by a transaction (other than a split completion transaction). If a target terminates a transaction with a split response, the completer becomes the initiator of the subsequent split completion.

Context synchronization. An operation that ensures that all instructions in execution complete past the point where they can produce an *interrupt*, that all instructions in execution complete in the context in which they began execution, and that all subsequent instructions are *fetch*ed and executed in the new context. Context synchronization may result from executing specific instructions (such as **isync** or **rfi**) or when certain events occur (such as an interrupt).

Copy-back operation. A cache operation in which a cache line is copied back to memory to enforce cache coherency. Copy-back operations consist of snoop push-out operations and cache cast-out operations.

D **Direct-mapped cache.** A cache in which each main memory address can appear in only one location within the cache; operates more quickly when the memory request is a cache hit.

Double data rate. Memory that allows data transfers at the start and end of a clock cycle, thereby doubling the data rate.

E **Exception.** A condition that, if enabled, can generate an interrupt.

Effective address (EA). The 32-bit address specified for a load, store, or an instruction fetch. This address is then submitted to the MMU for translation to either a *physical memory* or an I/O address.

Exclusive state. MEI state (E) in which only one caching device contains data that is also in system memory.

F **Fetch.** Retrieving instructions from either the cache or main memory and placing them into the instruction queue.

Flush. An operation that causes a cache block to be invalidated and the data, if modified, to be written to memory.

G **General-purpose register (GPR).** Any of the 32 registers in the general-purpose register file. These registers provide the source operands and destination results for all integer data manipulation instructions. Integer load

instructions move data from memory to GPRs and store instructions move data from GPRs to memory.

Guarded. The guarded attribute pertains to out-of-order execution. When a page is designated as guarded, instructions and data cannot be accessed out-of-order.

H **Harvard architecture.** An architectural model featuring separate caches and other memory management resources for instructions and data.

I **IEEE Std. 754.** A standard written by the Institute of Electrical and Electronics Engineers that defines operations and representations of binary floating-point numbers.

Illegal instructions. A class of instructions that are not implemented for a particular PowerPC processor. These include instructions not defined by the PowerPC architecture. In addition, for 32-bit implementations, instructions that are defined only for 64-bit implementations are considered to be illegal instructions. For 64-bit implementations instructions that are defined only for 32-bit implementations are considered to be illegal instructions.

Implementation. A particular processor that conforms to the PowerPC architecture, but may differ from other architecture-compliant implementations for example in design, feature set, and implementation of *optional* features. The PowerPC architecture has many different implementations.

Imprecise interrupt. A type of *synchronous exception* that is allowed not to adhere to the precise interrupt model (see *Precise exception*). The PowerPC architecture allows only floating-point interrupts to be handled imprecisely.

Integer unit. An execution unit in the core responsible for executing integer instructions.

In-order. An aspect of an operation that adheres to a sequential model. An operation is said to be performed in-order if, at the time that it is performed, it is known to be required by the sequential execution model.

Instruction latency. The total number of clock cycles necessary to execute an instruction and make ready the results of that instruction.

K **Kill.** An operation that causes a *cache block* to be invalidated without writing any modified data to memory.

-
- L**
- Latency.** The number of clock cycles necessary to execute an instruction and make ready the results of that execution for a subsequent instruction.
- L2 cache.** Level-2 cache. See *Secondary cache*.
- Least significant bit (lsb).** The bit of least value in an address, register, field, data element, or instruction encoding.
- Least significant byte (LSB).** The byte of least value in an address, register, data element, or instruction encoding.
- Little endian.** A byte-ordering method in memory where the address n of a word corresponds to the *least significant byte*. In an addressed memory word, the bytes are ordered (left to right) 3, 2, 1, 0, with 3 being the *most significant byte*. See *Big endian*.
-
- M**
- MESI (modified/exclusive/shared/invalid).** *Cache coherency* protocol used to manage caches on different devices that share a memory system. Note that the PowerPC architecture does not specify the implementation of a MESI protocol to ensure cache coherency.
- Memory access ordering.** The specific order in which the processor performs load and store memory accesses and the order in which those accesses complete.
- Memory-mapped accesses.** Accesses whose addresses use the page or block address translation mechanisms provided by the MMU and that occur externally with the bus protocol defined for memory.
- Memory coherency.** An aspect of caching in which it is ensured that an accurate view of memory is provided to all devices that share system memory.
- Memory consistency.** Refers to agreement of levels of memory with respect to a single processor and system memory (for example, on-chip cache, secondary cache, and system memory).
- Memory management unit (MMU).** The functional unit that is capable of translating an *effective (logical) address* to a physical address, providing protection mechanisms, and defining caching methods.
- Modified state.** MESI state (M) in which one, and only one, caching device has the valid data for that address. The data at this address in external memory is not valid.
- Most significant bit (msb).** The highest-order bit in an address, registers, data element, or instruction encoding.

Most significant byte (MSB). The highest-order byte in an address, registers, data element, or instruction encoding.

N

NaN. An abbreviation for not a number; a symbolic entity encoded in floating-point format. There are two types of NaNs—signaling NaNs and quiet NaNs.

No-op. No-operation. A single-cycle operation that does not affect registers or generate bus activity.

P

Page. A region in memory. The OEA defines a page as a 4-Kbyte area of memory aligned on a 4-Kbyte boundary.

Page access history bits. The *changed* and *referenced* bits in the PTE keep track of the access history within the page. The referenced bit is set by the MMU whenever the page is accessed for a read or write operation. The changed bit is set when the page is stored into. See *Changed bit* and *Referenced bit*.

Page fault. A page fault is a condition that occurs when the processor attempts to access a memory location that does not reside within a *page* not currently resident in *physical memory*. On PowerPC processors, a page fault exception condition occurs when a matching, valid *page table entry* (PTE[V] = 1) cannot be located.

Page table. A table in memory is comprised of *page table entries*, or PTEs. It is further organized into eight PTEs per PTEG (page table entry group). The number of PTEGs in the page table depends on the size of the page table (as specified in the SDR1 register).

Page table entry (PTE). Data structures containing information used to translate *effective address* to physical address on a 4-Kbyte page basis. A PTE consists of 8 bytes of information in a 32-bit processor and 16 bytes of information in a 64-bit processor.

Physical memory. The actual memory that can be accessed through the system's memory bus.

Pipelining. A technique that breaks operations, such as instruction processing or bus transactions, into smaller distinct stages or tenures (respectively) so that a subsequent operation can begin before the previous one has completed.

Precise interrupts. A category of interrupt for which the pipeline can be stopped so instructions that preceded the faulting instruction can complete and subsequent instructions can be flushed and redispached after interrupt handling has completed. See *Imprecise exceptions*.

Primary opcode. The most-significant 6 bits (bits 0–5) of the instruction encoding that identifies the type of instruction.

Program order. The order of instructions in an executing program. More specifically, this term is used to refer to the original order in which program instructions are fetched into the instruction queue from the cache.

Protection boundary. A boundary between *protection domains*.

Protection domain. A protection domain is a segment, a virtual page, a BAT area, or a range of unmapped effective addresses. It is defined only when the appropriate relocate bit in the MSR (IR or DR) is 1.

Q

Quad word. A group of 16 contiguous locations starting at an address divisible by 16.

Quiesce. To come to rest. The processor is said to quiesce when an interrupt is taken or a **sync** instruction is executed. The instruction stream is stopped at the decode stage and executing instructions are allowed to complete to create a controlled context for instructions that may be affected by out-of-order, parallel execution. See *Context synchronization*.

R

rA. The **rA** instruction field is used to specify a GPR to be used as a source or destination.

rB. The **rB** instruction field is used to specify a GPR to be used as a source.

rD. The **rD** instruction field is used to specify a GPR to be used as a destination.

rS. The **rS** instruction field is used to specify a GPR to be used as a source.

Record bit. Bit 31 (or the **Rc** bit) in the instruction encoding. When it is set, updates the condition register (CR) to reflect the result of the operation.

Referenced bit. One of two *page history bits* found in each *page table entry*. The processor sets the *referenced bit* whenever the page is accessed for a read or write. See also *Page access history bits*.

Reservation. The processor establishes a reservation on a *cache block* of memory space when it executes an **lwarx** instruction to read a memory semaphore into a GPR.

Reservation station. A buffer between the dispatch and execute stages that allows instructions to be dispatched even though the results of instructions on which the dispatched instruction may depend are not available.

RISC (reduced instruction set computing). An *architecture* characterized by fixed-length instructions with nonoverlapping functionality and by a separate set of load and store instructions that perform memory accesses.

S

Secondary cache. A cache memory that is typically larger and has a longer access time than the primary cache. A secondary cache may be shared by multiple devices. Also referred to as L2, or level-2, cache.

Sequence. In PCI-X, a sequence is one or more transactions associated with carrying out a single logical transfer by a requester. Each transaction in the same sequence carries the same unique sequence ID.

Set (*v*). To write a nonzero value to a bit or bit field; the opposite of *clear*. The term ‘set’ may also be used to generally describe the updating of a bit or bit field.

Set (*n*). A subdivision of a *cache*. Cacheable data can be stored in a given location in one of the sets, typically corresponding to its lower-order address bits. Because several memory locations can map to the same location, cached data is typically placed in the set whose *cache block* corresponding to that address was used least recently. See *Set associative*.

Set associative. Aspect of cache organization in which the cache space is divided into sections, called *sets*. The cache controller associates a particular main memory address with the contents of a particular set, or region, within the cache.

Slave. The device addressed by a master device. The slave is identified in the address tenure and is responsible for supplying or latching the requested data for the master during the data tenure.

Snooping. Monitoring addresses driven by a bus master to detect the need for coherency actions.

Snoop push. Response to a snooped transaction that hits a modified cache block. The cache block is written to memory and made available to the snooping device.

Stall. An occurrence when an instruction cannot proceed to the next stage.

Sticky bit. A bit that when *set* must be cleared explicitly.

Superscalar machine. A machine that can issue multiple instructions concurrently from a conventional linear instruction stream.

Supervisor mode. The privileged operation state of a processor. In supervisor mode, software, typically the operating system, can access all control registers and can access the supervisor memory space, among other privileged operations.

Synchronization. A process to ensure that operations occur strictly *in order*. See *Context synchronization*.

Synchronous interrupt. An *interrupt* that is generated by the execution of a particular instruction or instruction sequence. There are two types of synchronous interrupts, *precise* and *imprecise*.

System memory. The physical memory available to a processor.

T

Time-division multiplex (TDM). A single serial channel used by several channels taking turns.

Tenure. The period of bus mastership. There can be separate address bus tenures and data bus tenures.

TLB (translation lookaside buffer). A cache that holds recently-used *page table entries*.

Throughput. The measure of the number of instructions that are processed per clock cycle.

Transaction. A complete exchange between two bus devices. A transaction is typically comprised of an address tenure and one or more data tenures, which may overlap or occur separately from the address tenure. A transaction may be minimally comprised of an address tenure only.

Transfer termination. Signal that refers to both signals that acknowledge the transfer of individual beats (of both single-beat transfer and individual beats of a burst transfer) and to signals that mark the end of the tenure.

U

User mode. The operating state of a processor used typically by application software. In user mode, software can access only certain control registers and can access only user memory space. No privileged operations can be performed. Also referred to as problem state.

V

Virtual address. An intermediate address used in the translation of an *effective address* to a physical address.

Virtual memory. The address space created using the memory management facilities of the processor. Program access to *virtual memory* is possible only when it coincides with *physical memory*.

W

Way. A location in the cache that holds a cache block, its tags, and status bits.

Word. A 32-bit data element.

Write-back. A cache memory update policy in which processor write cycles are directly written only to the cache. External memory is updated only indirectly, for example, when a modified cache block is *cast out* to make room for newer data.

Write-through. A cache memory update policy in which all processor write cycles are written to both the cache and memory.

Index

A

Additional BATn registers, 5-24

Address

- mapping examples, PTEG, 5-53
- translation, 5-1

Address bus

- MPX bus mode
 - transfer attribute signals, 8-11

Address translation

- 32-bit physical address, 5-11
- 36-bit physical address, 5-12

Address translation, see also Memory management unit

Addressing

- extended block physical address, 36-bit physical address, 5-26
- extended, 36-bit physical address, 5-6
- modes, 2-53

Alignment

- exception, 4-23
- misaligned accesses, 2-49

AltiVec technology

- addition to machine state register, 7-2
- cache
 - LRU instruction support, 3-40
 - overview, 7-15
- denormalized numbers, 7-10
- exceptions
 - assist, 4-2, 4-30
 - DSI, 4-2, 4-20
 - overview, 7-15
 - unavailable, 4-2
- instruction timing
 - data stream
 - termination, 7-8
 - touch instructions and **sync**, 7-8
 - touch instructions and **tlbsync**, 7-8
 - dss** and **dssall** instructions, 7-10
 - dst** vs. **dstt** (differences), 7-9
 - dstst** vs. **dststt** (differences), 7-9
 - execution latency, 6-45
 - general, 7-16
 - LRU instructions, 7-4
 - overview, 6-1, 7-1
 - pipeline stalls (data stream instructions), 7-7
 - speculative execution (data stream instructions), 7-7
 - static data stream touch instructions, 7-8

- stream engine tags, 7-7
- transient data stream touch instructions, 7-8
- VALU execution timing, 6-34
- vector FP compare, min, max
 - in Java mode, 7-12
 - in non-Java mode, 7-11
- VFPU execution timing, 6-34
- VIU1 execution timing, 6-34
- VIU2 execution timing, 6-34
- VPU execution timing, 6-34

instructions, 6-33

- denormalization, 7-10
- instruction set, 2-91, 6-33, 7-4
- round-to-integer in Java mode, 7-13
- round-to-integer in non-Java mode, 7-12

Java mode, 7-3, 7-10

- memory management unit, 5-2, 7-16
- memory operations, 10-17
- NaNs, 7-10
- performance monitor, 10-1
- permute unit (VPU), 6-4
- programming model, 7-1
- register file structure, 7-2
- simple integer unit (VIU1, VIU2), 6-4
- transient instructions, 7-5
- unavailable exception, 4-26
- zeros, 7-10

Arithmetic instructions

- floating-point, A-48
- integer, 2-91, A-45
- vector floating-point, 2-96
- vector integer, 2-92

B

BAMR (breakpoint address mask register), 2-43, 10-9

BATL_n registers, 5-24

BAT_n (block address translation registers)

- extended addressing
 - additional BAT registers (e600 core), 5-24
 - BATL_n, 5-24
 - extended block size, 5-27
 - implementation of BAT array, 5-24
 - initialization, 5-23
 - summary, 5-29
 - WIMG bits, 5-25

Block - see cache block

- Block address translation
 - flow, 5-15
 - generation of physical addresses, 5-26
 - selection, 5-10
 - size options, 5-25
 - summary, 5-29
 - Block diagram, 8-7
 - Block size, 5-2
 - BO field, branch instruction, 6-27
 - Boundedly undefined, definition, 2-51
 - Branch
 - considerations, 6-53
 - execute, 6-7
 - fall-through, 6-24
 - folding, 6-24, 6-58
 - link stack, 10-15, 10-22, 10-25
 - loop example, 6-53
 - prediction, 10-27
 - processing, 10-22, 10-24
 - resolution, 10-22, 10-25, 10-27
 - resolution definition, 6-2
 - Branch instructions
 - address calculation, 2-73
 - condition register logical, 2-74, A-53
 - description, A-52
 - list, 2-73
 - list of instructions, A-52
 - system linkage, 2-74, 2-82, A-53
 - trap, 2-74, A-53
 - Branch prediction
 - definition, 6-2
 - mispredict example, 6-53
 - static, 6-27
 - Branch processing unit (BPU)
 - branch instruction timing, 6-29
 - execution timing, 6-23
 - Branch target instruction cache (BTIC), 3-37
 - Branch-taken bubble, example, 6-52
 - BTIC, 3-37
 - BTIC (branch target instruction cache), 6-11, 10-22, 10-25
 - Byte ordering
 - default, 2-91
 - support, 2-53
- C**
- Cache
 - AltiVec technology
 - LRU instruction support, 3-40
 - overview, 7-15
 - arbitration, 6-15
 - atomic memory references, 3-26
 - block, definition, 3-10
 - castout, 3-7
 - coherency
 - general, 3-12
 - support, 3-14
 - committed store queue (CSQ), definition, 3-6
 - control
 - bus operations, 3-59
 - dcbi**, 2-87
 - dcbt**, 2-80
 - instructions, 3-27, 3-31
 - overview, 3-27
 - core bus interface, 3-58
 - data cache
 - block fill operations, 3-36
 - block push operation, 3-38
 - cache block replacement selection, 3-38
 - configuration, 3-10, 3-45
 - configuration diagram, 3-10
 - dcba**, 3-34
 - dcbf**, 3-34
 - dcbi**, 3-35
 - dcbst**, 3-33
 - dcbt**, 3-31
 - dcbstst**, 3-32
 - dcbz**, 3-33
 - locking using HID0, 3-28
 - operation, 8-9
 - snooping, 3-62
 - status bits, 3-15
 - store hit, 3-38
 - data intervention, 3-17
 - data way locking, 2-34
 - enforcing store ordering, 3-26
 - finished store queue (FSQ), definition, 3-6
 - flushing, 3-48
 - guarded memory bit (G bit), 3-12
 - inhibited accesses (I bit), 3-12
 - instruction cache
 - block fill operations, 3-36
 - cache block replacement selection, 3-38
 - enabling/disabling with HID0, 3-29
 - icbi** instruction, 3-35
 - locking using HID0, 3-29
 - organization, diagram, 3-11
 - L1 caches
 - castout queue (LCQ), 3-7
 - control, 3-27
 - features list, 3-1
 - flushing, 3-41
 - invalidation, 3-41
 - misses, 3-8
 - operation, 8-9

- organization, data cache, 3-10
- organization, instruction cache, 3-11
- service queues, 3-8
- L2 cache
 - allocation, 3-53
 - block, 3-9
 - cache line replacement algorithms, 3-54
 - cache miss and reload operations, 3-53
 - considerations, 6-20
 - control, 3-46
 - ECC, *see* ECC (error correction code)
 - features list, 3-3
 - instruction interactions, 3-51
 - invalidation, 3-47
 - L2CR parameters, 3-46
 - memory configuration, 3-46
 - operation, 3-52, 8-10
 - operations caused by L1 requests, 3-55
 - parity error checking, 3-47
 - parity error reporting, 3-49
 - prefetch engines, 3-49
 - register descriptions, 2-22–2-33
 - replacement algorithm, 3-49
 - store data merging, 3-54
- load and store operations, 3-24
- load miss, 3-7
- load miss queue (LMQ), 3-7
- load ordering, 3-26
- load/store operations, processor initiated, 3-24
- load/store unit (LSU), 3-6
 - store queues, 3-6
- loads and LSU, 3-6
- LSU load miss, castout, and push queues, 3-7
- management instructions, 2-103, 7-6, A-54
- memory
 - access and sequential consistency, 3-25
 - coherency, 3-12, 6-36
 - subsystem, 3-8
- memory/cache access attributes, 3-12
- miss allocation, 3-37, 3-53
- model timing coherency, 6-36
- operations
 - data cache block
 - fill, 3-36
 - push, 3-38
 - instruction cache block fill, 3-36
 - load/store, processor initiated, 3-24
- out-of-order accesses to guarded memory, 3-13
- overview, 3-1
- PLRU replacement, 3-39
- push queues, 3-7
- reservation snooping, 3-23
- snoop response, 3-16
- store
 - gathering/merging, 3-7
 - miss merging, 3-38
- store data merging, 3-54
- store ordering, 3-26
- timing
 - data cache hit, 6-15
 - data cache miss, 6-18
 - instruction cache
 - and L2 cache hit, 6-20
 - throttling, 2-38, 9-9
- transaction types, 3-17
- transfer attribute signals, 3-60
- transient data and different coherency states
 - different cache states, 3-16
- WIMG bits, 3-12
- write-through mode (W bit), 3-12
- Cache management instructions, 2-103, 7-6, A-54
- Cache/memory subsystem block diagram, 3-5
- Care, 3-13
- Castout queue (LCQ), 3-7
- Changed (C) bit maintenance recording, 5-13–5-37
- Checkstop
 - operation, 8-5
 - signal, 8-5
 - state exception, 4-20
- ci (cache inhibit), 8-12
- ckstp_in/ckstp_out (checkstop input/output) signals, 8-5
- Classes of instructions, 2-51
- Committed store queue (CSQ), 6-68
 - definition, 3-6
- Compare instructions
 - floating-point, A-49
 - integer, A-46
 - vector floating-point, 2-97
 - vector integer, 2-94
- Completion
 - considerations, 6-22
 - definition, 6-2
 - of instruction, 6-8
 - resource requirements, 6-63
- Configuration signals sampled at reset, 8-4
- Context synchronization, 2-54
- Control flow, 2-99
- Control registers synchronization requirements, 2-55
- Conventions, xxxvii, xli, 6-1
- Core bus interface
 - 60x bus mode
 - bus snooping, 3-2
 - cache operation, 3-58
 - caches and bus snooping, 3-64, 3-65

- checkstop operation, 8-5
- direct-store accesses, 8-11
- memory accesses, MPX bus mode, 8-10
- MSSCR0, 8-10
- MSSSR0, 8-10
- reset signal interactions, 8-5
- Core interface
 - transfer type
 - encodings, 8-11
- Core interface block, 3-9
- CR (condition register)
 - CR6 bit settings for vector integer compare instructions, 2-94
 - execution latencies, 6-38
 - logical instructions, 2-74, A-53
- CSQ (committed store queue)
 - 5-entry, 6-68
 - definition, 3-6

D

- Data address breakpoint and exceptions, 4-21
- Data bus
 - MPX bus mode
 - intervention, 2-21
- Data cache
 - block fill operations, 3-36
 - block push operation, 3-38
 - configuration, 3-10, 3-45
 - locking, 3-28
 - operation, 8-9
 - organization diagram, 3-10, 3-45
- Data organization in memory, 2-48
- Data stream
 - prefetching and exceptions, 4-14
 - touch instructions
 - overview, 7-6
 - sync**, 7-8
 - termination, 7-8
 - tlbsync**, 7-8
- Data TLB miss-
 - on-load, 4-28
 - on-store, 4-28
- Data transfers
 - operand conventions, 2-48
- dcba**, 3-34
- dcbf**, 3-34
- dcbi**, 2-87, 3-35
- dcbst**, 3-33
- dcbt**, 2-80, 3-31
- dcbstt**, 3-32
- dcbz**, 3-33
- Decrementer exception, 4-24

- Defined instruction class, 2-51
- Direct-store accesses, 8-11
- Dispatch
 - considerations, 6-22
 - definition, 6-2
 - notation, 6-7
 - unit resource requirements, 6-58
- Doze mode, 9-2
 - see also* Power management
- DSI exception, 4-20
- DTLB organization, 5-39
- Dynamic branch prediction, 6-12

E

- e600
 - register set summary, 2-3
- e600 core
 - block diagram, overview, 8-7
- ECC (error correction code), 3-49
 - enabling or disabling, 3-50
 - error control and capture, 3-50
 - error control and capture registers, 2-26–2-33
 - error injection, 3-51
 - error injection registers, 2-25–2-26
 - error reporting, 3-51
- Effective address (EA), 5-1, 6-31
- Effective address calculation
 - branches, 2-54
 - loads and stores, 2-54, 2-66, 2-70
 - translation, 5-4
- ei**, 2-79
- Error correction code, *see* ECC (error correction code)
- Events
 - counting, 10-12
 - MMCRn registers, 10-12
 - PMC1, 10-13
 - PMC2, 10-19
 - PMC3, 10-23
 - PMC4, 10-25
 - PMC5, 10-27
 - PMC6, 10-28
 - PMCn registers selection, 10-13–10-28
- Exception handlers code for MMU page table search
 - software example, 5-71
- Exceptions
 - alignment, 4-23
 - Altivec
 - assist, 4-2, 4-30
 - disabled, 4-2
 - DSI, 4-2
 - technology overview, 7-15
 - unavailable, 4-26

- checkstop state, 4-20
- classification, 4-3
- conditions causing, 4-3
- data address breakpoint facility, 4-21
- data stream prefetching, 4-14
- data TLB miss-
 - on-load, 4-28
 - on-store DTLB, 4-28
- decrementer, 4-24
- definitions, 4-14
- DSI, 4-20
- enabling and disabling, 4-12
- external interrupt, 4-22
- floating point
 - assist, 4-25
 - unavailable, 4-24
- instruction address breakpoint, 4-28
- instruction TLB miss, 4-26, 4-27
- instruction-related exceptions, 2-58
- ISI, 4-21
- machine check, 4-16
- microprocessor, 4-3
- MMU
 - conditions, 5-20
 - summary, 5-18
- mode, 4-2
- MSR settings, 4-14
- overview, 4-3
- performance monitor, 4-25, 10-2
- prefix (IP) bit, 4-15
- priorities, 4-5
- processing
 - general, 4-8
 - steps, 4-12
- program, 4-23
- recognition, 4-5
- register settings
 - MSR, 2-9, 4-9, 4-14
 - SRRn, 2-12, 4-8
- reset, 4-15
- returning from handler, 4-13
- system call, 4-24
- system management interrupt, 4-29
- terminology, 4-2
- trace, 4-24
- translation conditions, 5-18

Execution

- instructions, 6-8
- synchronization, 2-58

Execution timing, FPU, 6-30

Execution units

- Altivec

- permute unit (VPU), 6-4
- simple integer unit (VIU1), 6-4
- vector complex integer unit (VIU2), 6-5
- vector floating-point unit (VFPU), 6-5
- multiple-cycle IU (IU2), 6-4
- timing
 - examples, 6-23
 - LSU, 6-31

External control instructions, A-57

External interrupt exception, 4-22

F

Fall-through folding, definition, 6-2

Fetch

- alignment example, 6-51
- definition, 6-2
- examples, 6-50
- instruction timing, 6-7

Fetch/branch considerations, 6-50

Finish definition, 6-2

Finished store queue (FSQ)

- general, 6-68

Finished store queue (FSQ), definition, 3-6

FIQ (floating-point issue queue), 6-63

Floating-point model

- arithmetic instructions, 2-63, A-48
- assist exceptions, 4-25
- compare instructions, 2-64, A-49
- FE0/FE1 bits, 2-11, 4-11
- FP move instructions, A-52
- FPSCR instructions, 2-64
- IEEE-754 compatibility, 2-48
- instructions, A-49
- load instructions, A-52
- multiply-add instructions, 2-63, A-48
- operands, 2-49
- rounding/conversion instructions, 2-64, A-49
- store instructions, 2-71, A-52
- unavailable exception, 4-24
- vector
 - compare instructions, 2-97
 - FP arithmetic instructions, 2-96
 - FP multiply-add, 2-96
 - FP rounding/conversion instructions, 2-96

Flushing L1 and L2 caches, 3-48

Folding definition, 6-2

FPSCR (floating-point status and control register)

- instructions, 2-64, A-49
- NI bit, 2-49

FPU (floating-point unit)

- performance exceptions, 6-31

FPU (floating-point unit) execution

- latencies, 6-41
- timing, 6-30
- FSQ (finished store queue)
 - definition, 3-6
 - transfers, 6-68

G

- GIQ (GPR issue queue), 6-61
- GPR issue queue (GIQ), 6-61

H

- Halted state, *see* Power management, core states
- Hashed page tables, 5-46
- Hashing functions
 - page table
 - primary PTEG, 5-49, 5-54
 - secondary PTEG, 5-49, 5-55
- HIDn (hardware implementation-dependent) registers
 - HID0
 - bit descriptions, 2-14
 - cache control parameters, 3-28
 - data cache locking, 3-28
 - instruction cache
 - locking, 3-29
 - instruction cache enabling/disabling, 3-29
 - XAEN (Extended addressing) bit, 2-15
 - HID1
 - bit descriptions, 2-18
 - PLL configuration, 2-20
- hreset (hard reset) signal, 8-5

I

- IABR (instruction address breakpoint register), 2-35, 10-16
- icbi** instruction, 3-35
- ICTC (instruction cache throttling control) register, 2-38
- ICTRL (instruction cache and interrupt control) register, 2-33
- IEEE 1149.1-compliant interface, 8-6
- Illegal instruction class, 2-52
- Implementation-specific instructions, 2-88
- Instruction address breakpoint exception, 4-28
- Instruction and data cache registers, 2-23
- Instruction cache
 - block fill operations, 3-36
 - enabling/disabling, 3-29
 - locking, 3-29
 - organization diagram, 3-11
 - throttling, 2-38, 9-9
- Instruction fetch
 - stages, 6-6
 - timing, 6-15

- Instruction pipeline stages
 - complete, 6-7
 - decode/dispatch, 6-6
 - execute, 6-7
 - general, 6-6
 - instruction fetch, 6-6
 - issue queues (FIQ, VIQ, GIQ), 6-6
 - write-back, 6-7
- Instruction timing
 - AltiVec technology
 - execution latency, 6-45
 - instructions, 6-33
 - overview, 6-1, 6-34
 - timing, 7-16
 - VFPU execution timing, 6-34
 - VIU1 execution timing, 6-34
 - VIU2 execution timing, 6-34
 - VPU execution timing, 6-34
 - branch execute, 6-7
 - completion of instruction, 6-8
 - CR execution latencies, 6-38
 - dispatch, 6-7
 - examples
 - cache hit, 6-16
 - cache miss, 6-19
 - execute, 6-8
 - execution unit, 6-23
 - fetch, 6-7
 - FPU execution latencies, 6-41
 - instruction flow, 6-10
 - issue, 6-8
 - LSU execution latencies, 6-42
 - memory coherency and the cache, 6-36
 - memory performance considerations, 6-36
 - overview, 6-3
 - terminology, 6-1
 - write-back, 6-8
- Instructions
 - addressing modes, 2-53
 - AltiVec
 - cache management, 2-103
 - execution latency, 6-45
 - general, 6-33
 - instruction set, 7-4
 - transient instructions, 7-5
 - user-level instructions, 2-103
 - boundedly undefined, 2-51
 - branch, 6-24, 6-25, A-52
 - address calculation, 2-73
 - BO field, 6-27
 - fetch/branch considerations, 6-50
 - predicting and resolution, 6-26

- cache
 - management instructions, 2-103, 3-31, 7-6, A-54
- cache throttling, 2-38, 9-9
- classes of instructions, 2-51
- condition register logical, 2-74, A-53
- context synchronization, 2-54
- defined instruction class, 2-51
- effective address calculation, 2-54
- exceptions, 2-58
- execution
 - serialization, 6-23
 - synchronization, 2-58
- execution latencies, 6-38
- external control, A-57
- floating-point
 - arithmetic, 2-63, A-48
 - compare, 2-64, A-49
 - estimate instructions, A-57
 - FPSCR instructions, A-49
 - instructions execution latencies, 6-41
 - load instructions, A-52
 - move, 2-65
 - move instructions, A-52
 - multiply-add, 2-63, A-48
 - rounding and conversion, 2-64, A-49
 - status and control register, 2-64
 - store instructions, A-52
- flow
 - control, 2-99
 - diagram, 6-14
- illegal instruction class, 2-52
- implementation-specific, 2-88
- instructions not implemented, B-1
- integer
 - arithmetic, 2-59, 2-91, A-45
 - compare, 2-60, A-46
 - load, A-50
 - load/store multiple, 2-69, A-51
 - load/store string, A-51
 - load/store with byte reverse, A-51
 - logical, 2-61, 2-91, A-46
 - rotate and shift, 2-61, A-47
 - store, 2-68, A-50
- isync**, 4-13
- latency summary, 6-37
- load and store
 - address generation
 - floating-point, 2-70
 - integer, 2-66
 - byte reverse instructions, 2-69, A-51
 - execution latencies, 6-42
 - floating-point load, A-52
 - floating-point move, 2-65, A-52
 - floating-point store, 2-71, A-52
 - indirect integer load, 2-66
 - integer
 - load, A-50
 - multiple, 2-69
 - store, 2-68, A-50
 - memory synchronization, 2-77, 2-78, A-51
 - misalignment handling, 2-65
 - multiple instructions, A-51
 - string instructions, 2-69, A-51
 - vector load, 2-98
- LRU, 2-104
- memory control instructions, 2-79, 2-86
- memory synchronization instructions, 2-77, 2-78, A-51
- move to/from VSCR register, 2-102
- pipelining
 - DST instructions and the vector touch engine (VTE), 6-76
 - load hit, 6-69
 - load/store, 6-71
 - misalignment effects, 6-72
 - store hit, 6-70
 - store miss, 6-74
- PowerPC
 - instruction list by functional categories, A-45
 - instruction list sorted by mnemonic, A-1
 - instruction list sorted by opcode, A-34
 - OEA instructions, 2-82
 - UISA instructions, 2-59
- processor control, 2-75, 2-78, 2-82, A-53
- refetch serialization, 6-23
- reserved instruction class, 2-53
- rfi**, 4-13
- segment register manipulation instructions, A-54
- serialization, 6-23, 6-63
- set summary, 2-50
- store serialization, 6-23
- stwcx.**, 4-14
- sync**, 4-13
- synchronization, 2-54
- system linkage, 2-74
- system linkage instructions, A-53
- system register instruction latencies, 6-37
- timing, 6-33
- TLB
 - management, A-54
 - miss exception, 4-27
- tlbld**, 2-88
- tlbli**, 2-88
- trap
 - general, 2-74

- instructions, A-53
 - vector
 - floating-point
 - arithmetic, 2-96
 - compare, 2-97
 - multiply-add, 2-96
 - rounding/conversion, 2-96
 - integer
 - arithmetic, 2-92, A-55
 - compare, 2-94
 - logical, 2-95
 - rotate/shift, 2-95
 - load (alignment support), 2-98
 - memory control, 2-103
 - merge, 2-100
 - pack, 2-99, A-57
 - permute, 2-101, A-58
 - select, 2-102, A-59
 - shift, A-59
 - splat, 2-101, A-58
 - status and control register, 2-102
 - store, 2-99
 - vector load, A-57
 - vrefp**, 7-14
 - int** (interrupt) signal, 8-5
 - Integer
 - arithmetic instructions, 2-59, 2-91, A-45
 - compare instructions, 2-60, A-46
 - indirect load instructions, 2-66
 - integer unit execution timing, 6-30
 - load instructions, A-50
 - logical instructions, 2-61, 2-91, A-46
 - rotate/shift instructions, 2-61, A-47
 - store gathering, 6-32
 - store instructions, 2-68, A-50
 - Interrupt handling
 - power management considerations, 9-8
 - Interrupt, external, 4-22
 - ISI exception, 4-21
 - Issue
 - definition, 6-2
 - illustration, 6-8
 - isync**, 2-79, 4-13
 - ITLB organization, 5-39
 - IU1 considerations, 6-64
 - IU2 considerations, 6-65
- L**
- L1 caches *see* Cache, L1 caches
 - L2 cache
 - implementation, 1-15
 - L2 cache *see* Cache, L2 cache
- L2CR (L2 cache control register)
 - general, 2-23
 - parameters, 3-46
 - parity checking, L2 cache, 3-47
 - Latency definition, 6-2
 - LCQ (L1 castout queue), 3-7
 - LDSTCR (load/store control register), 2-34
 - Link stack
 - example, 6-56
 - for branch indirect, 6-56
 - registers, 6-12
 - LMQ (load miss queue), 6-68
 - Load miss queue (LMQ), 3-7
 - Load/store
 - address generation, 2-66
 - byte reverse instructions, 2-69, A-51
 - floating-point
 - load instructions, 2-70, A-52
 - move instructions, 2-65, A-52
 - store instructions, 2-71
 - floating-point store instructions, A-52
 - integer
 - load instructions, A-50
 - store instructions, 2-68, A-50
 - integer load instructions, 2-66
 - load/store multiple instructions, 2-69, A-51
 - memory synchronization instructions, A-51
 - misalignment handling, 2-65
 - string instructions, 2-69, A-51
 - vector load instructions
 - alignment support, 2-98
 - general, 2-98
 - vector load/store instructions, 2-98
 - vector store instructions, 2-99
 - Load/store unit (LSU)
 - from cache, 3-6
 - general, 6-68
 - Logical address translation, 5-1
 - Logical instructions
 - integer, 2-91, A-46
 - vector integer, 2-95
 - Lookaside buffer management instructions, A-54
 - Loop example, 6-52
 - LRU (least recently used) instructions, 2-104, 6-33, 7-4
 - LSU
 - execution latencies, 6-42
 - execution timing, 6-31
 - load miss, castout, and push queues, 3-7
 - store queues, 3-6
- M**
- Machine check exception, 4-16

Memory

- accesses, MPX bus mode, 8-10
- cache interface, 3-8
- coherency, 3-12, 6-36
- control instructions
 - description, 2-79, 2-86
 - segment register manipulation, A-54
 - user-level cache, 2-103, 7-5
- Memory management unit
 - access protection, 5-1
 - address translation
 - flow, 5-15
 - mechanisms, 5-9, 5-14
 - address translation types, 32-bit physical addressing, 5-11
 - AltiVec technology overview, 5-2, 7-16
 - block address translation, 5-10, 5-15, 5-23
 - block diagram
 - 32-bit implementation, 5-5
 - DMMU (36-bit physical addressing), 5-8
 - DMMU (Extended Block Size and Additional BATs), 5-9
 - block size, 5-2
 - data stream touch instructions, 5-36, 5-37
 - effective address calculation, 5-4
 - exceptions
 - conditions, 5-18, 5-20
 - implementation-specific, 5-20
 - summary, 5-18
 - extended addressing, 2-37
 - extended BAT block size, 5-27
 - features summary, 5-3
 - hashing functions, 5-49
 - implementation-specific features, 5-2
 - instruction summary, 5-21
 - memory protection, 5-13
 - memory segment model, 5-31
 - no-execute protection, 5-16
 - organization, 5-4
 - overview, 5-2
 - page address translation, 5-10, 5-15, 5-44
 - page history status, 5-13, 5-35–5-37
 - PTEHI, PTELO registers, 2-36
 - real addressing mode
 - block address translation selection, 5-14
 - mechanisms, 5-14
 - support for real, 5-2
 - translation disabled, 5-23
 - referenced and changed bit scenarios, 5-37
 - register summary, 5-22
 - software table search operation, 5-65, 5-67
 - software table search registers, 2-36
 - table search operation

- conditions, 5-56
- example, 5-65
- hardware, 5-2
- TLB miss, 5-40
- updating history bits, 5-35

- tlbie**, 5-40

- tlbsync**, 5-42

- translation exception conditions, 5-18

- Memory segment model, page address translation

- overview, 5-32

- PTE definitions, 5-34

- segment descriptor definitions, 5-33

- Memory synchronization instructions, 2-77, 2-78, A-51

- MESI protocol and state transition, 3-18

- Misalignment

- in accesses, 2-49

- MMCR0 (monitor mode control register 0), 2-39

- MMCR1 (monitor mode control register 1), 2-42

- MMCR2 (monitor mode control register 2), 2-42

- MMCRn (monitor mode control registers), 2-39, 4-25, 10-4–10-9, 10-12

- MPX bus mode

- transfer type encodings, 8-11

- MSR (machine state register)

- bit settings, 2-9, 4-9

- FE0/FE1 bits, 2-11, 4-11

- IP bit, 4-15

- RI bit setting, 4-13

- settings due to exception, 4-14

- MSSCR0 (memory subsystem control register 0), 2-20, 3-49, 8-10

- MSSSR0 (memory subsystem status register 0), 2-22, 8-10

- Multiple-cycle IU (IU2), 6-4

- Multiple-precision shifts, 2-62

- Multiply-add instructions, 2-63, 2-96, A-48

N

- Nap mode, 9-2

- see also* Power management

O

- OEA instructions, 2-82

- Operand conventions, 2-48

- Operand placement, performance effects, 6-31

- Operating environment architecture (OEA)

- exception mechanism, 4-1

- memory management unit, 5-1

- registers, 2-8

- Operating environment architecture (OEA), xxxiv

- Operations

- bus operations caused by cache control instructions

- general, 3-59
- modified, 3-59
- not modified, 3-60
- data cache block
 - fill, 3-36
 - push, 3-38
- instruction cache block fill, 3-36
- response to snooped bus transactions, 3-63
- table search
 - hardware, 5-2
 - software, 5-2
 - TLB miss, 5-40
 - updating history bits, 5-35
- Optional instructions, A-75

P

- Page address translation
 - extended addressing
 - extended block size, 5-31
 - flow, 5-44
 - physical address generation, 5-33
 - PTE definitions, 5-34
 - overview, 5-32
 - selection, 5-10, 5-15, 5-16
 - TLB organization, 5-39
- Page history
 - recording, 5-35
 - status
 - dcbt** and **dcbtst** misses, 5-36
 - R and C bit recording, 5-13, 5-35–5-37
- Page table
 - example structures, 5-53
 - hashed, 5-46
 - hashing functions, 5-49, 5-54
 - PTE registers (PTEHI and PTELO), 2-36
 - PTEG
 - address generation examples, 5-53, 5-54
 - addresses, 5-50, 5-53
 - general, 5-53
 - search operation
 - software, 5-65
 - search operations
 - conditions, 5-56
 - hardware, 5-57
 - reads, 5-58
 - resources, 5-61
 - size, 5-48
 - updates, 5-60
- Parity error checking, L2 cache, 3-47
- Parity error reporting, L2 cache, 3-49
- Performance monitor, 1-20
 - AltiVec technology, 10-1

- counter registers, 10-10
- event counting, 10-12
- events, 10-12
- exception, 4-25, 10-2
- overview, 10-1
- registers, 10-3
- TBEE (timebase enable event) usage, 10-3
- uses for the performance monitor, 10-1
- Performance monitor registers, 2-39
- Physical address (PA)
 - generation of PTEG addresses, 5-50, 5-53
 - overview, 5-1
- Physical address generation
 - blocks, 5-26
- Pipeline
 - execution unit, 6-4
 - instruction timing definition, 6-2
 - stages, 6-9
 - superscalar diagram, 6-5
- PMCn (performance monitor counter registers), 2-44, 4-25, 10-10, 10-13–10-28
- Position-independent code example, 6-57
- Power management, 1-19
 - control bits, 9-6
 - device modes (doze, nap, and sleep), 9-2
 - dynamic power management, 9-6
 - interrupt recognition and servicing, 9-8
 - protocol between core and other device logic, 9-7
 - signals, 9-1
 - software considerations, 9-5
- Power states
 - deep sleep, 9-5
 - doze, 9-3
 - full-power, 9-3
 - nap, 9-4
 - sleep, 9-4
- Power-on reset settings, 2-45
- PowerPC architecture
 - byte ordering support, 2-53
 - instruction list, A-1, A-34, A-45
 - memory accesses and sequential consistency, 3-25
 - operating environment architecture (OEA), xxxiv
 - user instruction set architecture (UISA), xxxiii
 - virtual environment architecture (VEA), xxxiii
- Process switching, 4-13
- Processor control instructions, 2-75, 2-78, 2-82, A-53
- Program exception, 4-23
- Program order definition, 6-3
- Protection of memory areas
 - no-execute protection, 5-16
 - options available, 5-13
 - violations, 5-18

PTEGs (PTE groups)

- examples primary and secondary, 5-53
- generation of addresses, 5-50
- hashing, 5-49
- PTEHI (page table entry high register), 2-36, 5-64
- PTELO (page table entry low register), 2-36, 5-64
- PTEs (page table entries)
 - bit definitions, 2-37, 5-34, 5-65
 - extended addressing, 2-37
 - page history recording (PTE(R) and PTE(C)), 5-35
- PVR (processor version register), 2-8

Q

- qreq (quiescent request) signal, 8-5

R

- Real addressing mode, 5-22
- Real addressing mode (translation disabled)
 - 32-bit, 5-23
 - 36-bit, 5-23
 - data accesses, 5-14, 5-23
 - extended addressing, 5-23
 - instruction accesses, 5-14, 5-23
 - support, 5-2
- Referenced (R) bit maintenance recording, 5-13–5-37, 5-58
- Registers
 - addition to AltiVec machine state register, 7-2
 - AltiVec technology, 7-2
 - BATL_n, 5-24
 - data cache, 2-23
 - e600-specific, 2-13–2-45
 - implementation-specific
 - BAMR, 2-43, 10-9
 - HID0, 2-14
 - HID1, 2-18
 - IABR, 2-35
 - ICTC, 2-38
 - ICTRL, 2-33
 - L2CR, 2-23
 - LDSTCR, 2-34
 - MMCR_n, 2-39, 4-25, 10-4–10-9, 10-12
 - MSSCR0, 2-20, 8-10
 - MSSSR0, 2-22
 - PMC_n, 2-44, 4-25, 10-13–10-28
 - SIAR, 2-45, 4-25, 10-11
 - UMMCR_n, 2-41–2-43, 10-7–10-9
 - UPMC_n, 2-45, 10-11
 - USIAR, 2-45, 10-12
 - instruction, 2-23
 - L2 cache
 - error control and capture, 2-26–2-33

- error injection, 2-25–2-26
- MSSSR0, 8-10
- overview, 2-2
- page table entry, 2-36
- performance monitor
 - counter, 10-10
 - overview, 2-39, 10-3
- reset settings, 2-45
- segment, 5-33
- segment updates, 5-61
- software table search, 2-36
- SPR encodings, 2-85
- SPR encodings (e600-defined registers), 2-76
- supervisor-level
 - BAMR, 2-43, 10-9
 - BATs, 5-24
 - DMISS and IMISS, 5-63
 - HID0, 2-14
 - HID1, 2-18
 - IABR, 2-35
 - ICTC, 2-38
 - ICTRL, 2-33
 - L2CAPTDATAHI, 2-26
 - L2CAPTDATALO, 2-27
 - L2CAPTECC, 2-27
 - L2CR, 2-23
 - L2ERRADDR, 2-31
 - L2ERRATTR, 2-30
 - L2ERRCTL, 2-32
 - L2ERRDET, 2-28
 - L2ERRDIS, 2-29
 - L2ERREADDR, 2-32
 - L2ERRINJCTL, 2-25
 - L2ERRINJHI, 2-25
 - L2ERRINJLO, 2-25
 - L2ERRINTEN, 2-30
 - LDSTCR, 2-34
 - MMCR_n, 2-39, 4-25, 10-4–10-9, 10-12
 - MSSCR0, 2-20, 8-10
 - MSSSR0, 2-22
 - performance monitor SPRs, 10-4
 - PMC_n, 2-44, 4-25, 10-13–10-28
 - PVR, 2-8, 2-9
 - SDR1, 2-12
 - SIAR, 2-45, 4-25, 10-11
 - TLBMISS, 2-36
- user performance monitor counter, 10-11
- user-level
 - performance monitor SPRs, 10-4
 - UMMCR_n, 2-41–2-43, 10-7–10-9
 - UPMC_n, 2-45, 10-11
 - USIAR, 2-45, 10-12

- VRSAVE, 7-4
- VSCR, 7-3
- VRn, 7-2
- Rename buffer, definition, 6-3
- Rename register operation, 6-22
- Reservation station, definition, 6-3
- Reserved instruction class, 2-53
- Reset
 - hard, 2-45
 - hreset signal, 8-5
 - settings at power-on, 2-45
 - sreset signal, 8-5
- Retirement, definition, 6-3
- rfi**, 4-13
- Rotate/shift instructions, 2-61, 2-95, A-47
- Rounding/conversion instructions, vector FP, 2-96

S

- SDR1 register
 - bit description for extended addressing, 2-12
 - definition, 5-46
 - format, 5-46
 - generation of PTEG addresses, 5-50, 5-53
- Segment register
 - descriptor definitions, 5-33
 - updates, 5-61
- Segmented memory model, *see* Memory management unit
- Serialization instructions, 6-23, 6-63
- Shift/rotate instructions, 2-61, A-47
- SIAR (sampled instruction address register), 2-45, 4-25, 10-11
- Signals
 - address transfer attribute, 8-11
 - \overline{ci} (cache inhibit), 8-12
 - configuration sampled at reset, 8-4
 - definition of groupings, 8-1
 - descriptions, 8-2
 - \overline{gbl} (global), 8-12
 - MPX bus mode
 - \overline{ci} , 8-12
 - \overline{ci} (cache inhibit), 8-12
 - \overline{gbl} (global), 8-12
 - overview, 8-7
 - TTn, 8-11
 - wt (write-through), 8-12
 - non-protocol specific
 - ckstp_in/ckstp_out, 8-5
 - hreset, 8-5
 - int, 8-5
 - JTAG interface, 8-6
 - qreq, 8-5
 - smi, 4-29
 - \overline{sreset} , 8-5
 - wt (write-through), 8-12
 - power management, 9-1
 - transfer encoding, 8-11
- Simplified mnemonics, 2-88
- Sleep mode, 9-2
 - see also* Power management
- smi (system management interrupt) signal, 4-29
- Snooping, 3-62, 8-7
- Software table search
 - exception handlers code example, 5-71
 - operation, 5-61
 - operation example, 5-65
- SPR encodings
 - e600-defined, 2-85
 - supervisor-level (PowerPC), 2-83
 - user-level, 2-76
- SPRGs (Special purpose registers), 5-65
- sreset (soft reset) signal, 8-5
- SRn (segment registers)
 - manipulation instructions, A-54
- SRRn (status save/restore registers)
 - key bit derivation, 5-63
- SRRn (status save/restore registers)
 - processing, 2-12, 4-8
- SRU (system register unit) execution latencies, 6-37
- Stage, definition, 6-3
- Stall, definition, 6-3
- Static branch prediction, 6-12, 6-27
- Static versus dynamic prediction, 6-55
- Stopped state, *see* Power management, core states
- Store gathering/merging, 3-7
- stwcx.**, 4-14
- Superscalar, definition, 6-3
- SVR (system version register), 2-9
- Switching process, 4-13
- sync**, 4-13
- Synchronization
 - context/execution synchronization, 2-54
 - control registers requirements, 2-55
 - execution of **rfi**, 4-13
 - memory synchronization instructions, 2-77, 2-78, A-51
- System
 - call exception, 4-24
 - interface
 - general, 1-17
 - linkage instructions, 2-74, 2-82
 - management interrupt, 4-29
 - reset exception, 4-15
 - System bus interface
 - MPX bus mode
 - address transfer

- attributes, 8-11
- features, 8-7
- transfer type
 - signals, 8-11

System linkage instructions, A-53

T

Table search

- flow (primary and secondary), 5-58
- operations
 - example, 5-65
 - hashing functions, 5-49
 - SDR1 register, 5-46
 - software, 5-61, 5-65

tben (time base enable) signal, 2-78

Throughput, definition, 6-3

Timing, instruction

- BPU execution timing, 6-23
- branch timing example, 6-29
- cache hit, 6-16
- cache miss, 6-19
- execution unit, 6-23
- instruction dispatch, 6-22
- instruction flow, 6-10
- instruction scheduling guidelines, 6-49
- IU execution timing, 6-30
- latency summary, 6-37
- overview, 6-3
- rename register operation, 6-22
- stage definition, 6-3

TLB

- description, 5-38
- invalidation
 - description, 5-3, 5-40
 - tlbie** instruction, 5-40, 5-60
- LRU replacement, 5-40
- management instructions, 2-87, 3-14, A-54
- miss and table search operation, 5-40, 5-56
- miss exceptions, 4-26
- organization for ITLB and DTLB, 5-38

TLB miss exception

- DTLB miss-on-load, 4-28
- DTLB miss-on-store, 4-28
- ITLB miss, 4-27

tlbld, 2-89

tlbli, 2-90

TLBMIS (table miss register), 2-36, 5-63

Trace exception, 4-24

Transfer type encodings, MPX bus mode, 8-11

Transient caches, 7-5

Transient instructions, 3-1, 6-33, 7-5

Translation exception conditions, 5-18

Trap instructions, 2-74

TTn (transfer type) signals, 8-11

U

UISA (user instruction set architecture)

- registers, 2-8

UISA instructions, 2-59

UMMCRn (user monitor mode control registers), 2-41–2-43, 10-7–10-9

UPMCn (user performance monitor counter) registers, 2-45, 10-11

User performance monitor counter registers, 10-11

User instruction set architecture (UISA)

- description, xxxiii

USIAR (user sampled instruction address register), 2-45, 10-12

V

VALU (vector arithmetic logical unit), 6-34

Vector complex integer unit (VIU2), 6-5

Vector floating-point unit (VFPU), 6-5

Vector instructions

integer

- arithmetic, 2-92, A-55
- compare, 2-94
- logical, 2-95
- rotate/shift, 2-95

load, 2-98, A-57

load alignment support, 2-98, A-57

memory control, 2-103

merge, 2-100

pack, 2-99, A-57

permutation and formatting, 2-99

permute, 2-101, A-58

select, 2-102, A-59

shift, 2-102, A-59

splat, 2-101, A-58

status and control register, 2-102

store, 2-99

unpack, 2-100

Vector issue queue (VIQ), 6-62

Vector touch queue (VTE), 6-68

Vector unit considerations, 6-67

VFPU (vector floating-point unit), 6-34

VIQ (vector issue queue), 6-62

Virtual environment architecture (VEA), xxxiii

VIU1 (vector integer unit 1), 6-34

VIU2 (vector integer unit 2), 6-34

VPU (vector permute unit), 6-34

VRn (vector registers), 7-2

VRS (vector save/restore register), 7-4

VSCR (vector status and control register), 7-3

VTE (vector touch queue), 6-68

W

WIMG bits

default, 5-22

general, 3-12

in BAT register, 5-25

Write-back

definition, 6-3

general, 6-8

wt (write-through), 3-60,8-12



Overview	1
Registers	2
L1 and L2 Cache Operation	3
Interrupts	4
Memory Management	5
Instruction Timing	6
AltiVec Technology Implementation	7
Core Interface	8
Power and Thermal Management	9
Performance Monitor	10
e600 Core Instruction Set Listings	A
Instructions Not Implemented	B
Special-Purpose Registers	C
Revision History	D
Glossary	GLO
Index	IND



1 Overview

2 Registers

3 L1 and L2 Cache Operation

4 Interrupts

5 Memory Management

6 Instruction Timing

7 AltiVec Technology Implementation

8 Core Interface

9 Power and Thermal Management

10 Performance Monitor

A e600 Core Instruction Set Listings

B Instructions Not Implemented

C Special-Purpose Registers

D Revision History

GLO Glossary

IND Index