Cell Broadband Engine Programming Handbook

Version 1.0

April 19, 2006

© Copyright International Business Machines Corporation, Sony Computer Entertainment Incorporated, Toshiba Corporation 2006.

All Rights Reserved
Printed in the United States of America April 2006

The following are trademarks of International Business Machines Corporation in the United States or other countries, or both.

IBM	PowerPC	eServer
IBM Logo	PowerPC Architecture	ibm.com

Cell Broadband Engine is a trademark of Sony Computer Entertainment, Inc.

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Intel is a registered trademark of Intel Corporation or its subsidiaries in the United States and other countries.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Linux is a trademark of Linus Torvalds in the United States, other countries, or both.

Other company, product, and service names may be trademarks or service marks of others.

All information contained in this document is subject to change without notice. The products described in this document are NOT intended for use in applications such as implantation, life support, or other hazardous uses where malfunction could result in death, bodily injury, or catastrophic property damage. The information contained in this document does not affect or change IBM product specifications or warranties. Nothing in this document shall operate as an express or implied license or indemnity under the intellectual property rights of IBM or third parties. All information contained in this document was obtained in specific environments, and is presented as an illustration. The results obtained in other operating environments may vary.

THE INFORMATION CONTAINED IN THIS DOCUMENT IS PROVIDED ON AN “AS IS” BASIS. In no event will IBM be liable for damages arising directly or indirectly from any use of the information contained in this document.

IBM Systems and Technology Group
2070 Route 52, Bldg. 330
Hopewell Junction, NY 12533-6351

The IBM home page can be found at **ibm.com**
The IBM Semiconductor Solutions home page can be found at **ibm.com**/chips

Version 1.0
April 19, 2006

Contents

Contents **3**

List of Figures **19**

List of Tables **23**

Preface **29**

1. Overview of the Cell Broadband Engine Processor **33**
- 1.1 Background 34
  - 1.1.1 Motivation 34
  - 1.1.2 Power, Memory, and Frequency 35
  - 1.1.3 Scope of this Handbook 36
- 1.2 Hardware Environment 37
  - 1.2.1 The Processor Elements 37
  - 1.2.2 Element Interconnect Bus 37
  - 1.2.3 Memory Interface Controller 38
  - 1.2.4 Cell Broadband Engine Interface Unit 38
- 1.3 Programming Environment 39
  - 1.3.1 Instruction Sets 39
  - 1.3.2 Storage Domains and Interfaces 39
  - 1.3.3 Byte Ordering and Bit Numbering 41
  - 1.3.4 Runtime Environment 42

2. PowerPC Processor Element **45**
- 2.1 PowerPC Processor Unit 46
- 2.2 PowerPC Processor Storage Subsystem 48
- 2.3 PPE Registers 48
- 2.4 PowerPC Instructions 51
  - 2.4.1 Data Types 51
  - 2.4.2 Addressing Modes 51
  - 2.4.3 Instructions 52
- 2.5 Vector/SIMD Multimedia Extension Instructions 55
  - 2.5.1 SIMD Vectorization 55
  - 2.5.2 Data Types 57
  - 2.5.3 Addressing Modes 57
  - 2.5.4 Instruction Types 57
  - 2.5.5 Instructions 58
  - 2.5.6 Graphics Rounding Mode 59
- 2.6 Vector/SIMD Multimedia Extension C/C++ Language Intrinsics 59
  - 2.6.1 Vector Data Types 59
  - 2.6.2 Vector Literals 60
  - 2.6.3 Intrinsics 60

3. Synergistic Processor Elements **63**

3.1 Synergistic Processor Unit 63
3.1.1 Local Store 64
3.1.2 Register File 67
3.1.3 Execution Units 68
3.1.4 Floating-Point Support 68
3.2 Memory Flow Controller 70
3.2.1 Channels 72
3.2.2 Mailboxes and Signalling 72
3.2.3 MFC Commands and Command Queues 72
3.2.4 Direct Memory Access Controller 73
3.2.5 Synergistic Memory Management Unit 74
3.3 SPU Instruction Set 74
3.3.1 Data Types 74
3.3.2 Instructions 75
3.4 SPU C/C++ Language Intrinsics 76
3.4.1 Vector Data Types 76
3.4.2 Vector Literals 77
3.4.3 Intrinsics 77

4. Virtual Storage Environment 79
4.1 Introduction 79
4.2 PPE Memory Management 80
4.2.1 Memory Management Unit 81
4.2.2 Address-Translation Sequence 82
4.2.3 Enabling Address Translation 83
4.2.4 Effective-to-Real-Address Translation 83
4.2.5 Segmentation 85
4.2.6 Paging 87
4.2.7 Translation Lookaside Buffer 92
4.2.8 Real Addressing Mode 99
4.2.9 Effective Addresses in 32-Bit Mode 102
4.3 SPE Memory Management 102
4.3.1 Synergistic Memory Management Unit 102
4.3.2 Enabling Address Translation 103
4.3.3 Segmentation 104
4.3.4 Paging 107
4.3.5 Translation Lookaside Buffer 107
4.3.6 Real Addressing Mode 116
4.3.7 Exception Handling and Storage Protection 117

5. Memory Map 121
5.1 Introduction 121
5.1.1 Configuration-Ring Initialization 123
5.1.2 Allocated Regions of Memory 123
5.1.3 Reserved Regions of Memory 125
5.1.4 The Guarded Attribute 126
5.2 PPE Memory Map 126
5.2.1 PPE Memory-Mapped Registers 126
5.2.2 Predefined Real-Address Locations 126

5.3 SPE Memory Map 127
5.3.1 SPE Local-Store Memory Map 128
5.3.2 SPE Memory-Mapped Registers 129
5.4 BEI Memory-Mapped Registers 130
5.4.1 I/O 130

6. Cache Management 133
6.1 PPE Caches 133
6.1.1 Configuration 134
6.1.2 Overview of PPE Cache 134
6.1.3 L1 Caches 136
6.1.4 Branch History Table and Link Stack 141
6.1.5 L2 Cache 141
6.1.6 Instructions for Managing the L1 and L2 Caches 146
6.1.7 Effective-to-Real-Address Translation Arrays 150
6.1.8 Translation Lookaside Buffer 150
6.1.9 Instruction-Prefetch Queue Management 150
6.1.10 Load-Subunit Management 150
6.2 SPE Caches 151
6.2.1 Translation Lookaside Buffer 151
6.2.2 Atomic Unit and Cache 151
6.3 Replacement Management Tables 154
6.3.1 PPE TLB Replacement Management Table 154
6.3.2 PPE L2 Replacement Management Table 157
6.3.3 SPE TLB Replacement Management Table 158
6.4 I/O Address-Translation Caches 159

7. I/O Architecture 161
7.1 Overview 161
7.1.1 I/O Interfaces 161
7.1.2 System Configurations 162
7.1.3 I/O Addressing 164
7.2 Data and Access Types 165
7.2.1 Data Lengths and Alignments 165
7.2.2 Atomic Accesses 166
7.3 Registers and Data Structures 166
7.3.1 IOCmd Configuration Register 166
7.3.2 I/O Segment Table Origin Register 166
7.3.3 I/O Segment Table 169
7.3.4 I/O Page Table 171
7.3.5 IOC Base Address Registers 175
7.3.6 I/O Exception Status Register 176
7.4 I/O-Address Translation 176
7.4.1 Translation Overview 176
7.4.2 Translation Steps 179
7.5 I/O Exceptions 181
7.5.1 I/O Exception Causes 181
7.5.2 I/O Exception Status Register 182
7.5.3 I/O Exception Mask Register 182

7.5.4 I/O-Exception Response 182
7.6 I/O Address-Translation Caches 182
7.6.1 IOST Cache 182
7.6.2 IOPT Cache 184
7.7 I/O Storage Model 189
7.7.1 Memory Coherence 189
7.7.2 Storage-Access Ordering 190
7.7.3 I/O Accesses to Other I/O Units Through an IOIF 195
7.7.4 Examples 196

8. Resource Allocation Management 203
8.1 Introduction 203
8.2 Requesters 205
8.2.1 PPE and SPEs 205
8.2.2 I/O 205
8.3 Managed Resources 206
8.4 Tokens 207
8.4.1 Tokens Required for Single-CBE-Processor Systems 207
8.4.2 Operations Requiring No Token 210
8.4.3 Tokens Required for Multi-CBE-Processor Systems 211
8.5 Token Manager 212
8.5.1 Request Tracking 212
8.5.2 Token Granting 212
8.5.3 Unallocated RAG 214
8.5.4 High-Priority Token Requests 214
8.5.5 Memory Tokens 215
8.5.6 I/O Tokens 218
8.5.7 Unused Tokens 218
8.5.8 Memory Banks, IOIF Allocation Rates, and Unused Tokens 218
8.5.9 Token Request and Grant Example 219
8.5.10 Allocation Percentages 223
8.5.11 Efficient Determination of TKM Priority Register Values 224
8.5.12 Feedback from Resources to Token Manager 226
8.6 Configuration of PPE, SPEs, MIC, and IOC 227
8.6.1 Configuration Register Summary 227
8.6.2 SPE Address-Range Checking 229
8.7 Changing Resource-Management Registers with MMIO Stores 231
8.7.1 Changes to the RAID 231
8.7.2 Changing a Requestor’s Token-Request Enable 232
8.7.3 Changing a Requestor’s Address Map 233
8.7.4 Changing a Requestor’s Use of Multiple Tokens per Access 234
8.7.5 Changing Feedback to the TKM 234
8.7.6 Changing TKM Registers 234
8.8 Latency Between Token Requests and Token Grants 235
8.9 Hypervisor Interfaces 235

9. PPE Interrupts 237
9.1 Introduction 237
9.2 Summary of Interrupt Architecture 238

9.3 Interrupt Registers 241
9.4 Interrupt Handling 242
9.5 Interrupt Vectors and Definitions 243
9.5.1 System Reset Interrupt (Selectable or x‘00..00000100’) 245
9.5.2 Machine Check Interrupt (x‘00..00000200’) 246
9.5.3 Data Storage Interrupt (x‘00..00000300’) 248
9.5.4 Data Segment Interrupt (x‘00..00000380’) 249
9.5.5 Instruction Storage Interrupt (x‘00..00000400’) 250
9.5.6 Instruction Segment Interrupt (x‘00..00000480’) 251
9.5.7 External Interrupt (x‘00..00000500’) 251
9.5.8 Alignment Interrupt (x‘00..00000600’) 251
9.5.9 Program Interrupt (x‘00..00000700’) 252
9.5.10 Floating-Point Unavailable Interrupt (x‘00..00000800’) 254
9.5.11 Decrementer Interrupt (x‘00..00000900’) 254
9.5.12 Hypervisor Decrementer Interrupt (x‘00..00000980’) 254
9.5.13 System Call Interrupt (x‘00..00000C00’) 255
9.5.14 Trace Interrupt (x‘00..00000D00’) 256
9.5.15 VXU Unavailable Interrupt (x‘00..00000F20’) 256
9.5.16 System Error Interrupt (x‘00..00001200’) 257
9.5.17 Maintenance Interrupt (x‘00..00001600’) 258
9.5.18 Thermal Management Interrupt (x‘00..00001800’) 260
9.6 Direct External Interrupts 262
9.6.1 Interrupt Presentation 262
9.6.2 IIC Interrupt Registers 263
9.6.3 SPU and MFC Interrupts 268
9.6.4 Other External Interrupts 269
9.7 Mediated External Interrupts 273
9.7.1 Mediated External Interrupt Architecture 273
9.7.2 Mediated External Interrupt Implementation 275
9.8 SPU and MFC Interrupts Routed to the PPE 276
9.8.1 Interrupt Types and Classes 277
9.8.2 Interrupt Registers 278
9.8.3 Interrupt Definitions 282
9.8.4 Handling SPU and MFC Interrupts 285
9.9 Thread Targets for Interrupts 287
9.10 Interrupt Priorities 287
9.11 Interrupt Latencies 289
9.12 Machine State Register Settings Due to Interrupts 290
9.13 Interrupts and Hypervisor 291
9.14 Interrupts and Multithreading 291
9.15 Checkstop 291
9.16 Use of an External Interrupt Controller 291
9.17 Relationship Between CBE and PowerPC Interrupts 292

10. PPE Multithreading 295
10.1 Multithreading Guidelines 295
10.2 Thread Resources 297
10.2.1 Registers 297
10.2.2 Arrays, Queues, and Other Structures 298

10.2.3 Pipeline Sharing and Support for Multithreading 298
10.3 Thread States 301
10.3.1 Privilege States 301
10.3.2 Suspended or Enabled State 302
10.3.3 Blocked or Stalled State 302
10.4 Thread Control and Status Registers 302
10.4.1 Machine State Register (MSR) 303
10.4.2 Hardware Implementation Register 0 (HID0) 304
10.4.3 Logical Partition Control Register (LPCR) 305
10.4.4 Control Register (CTRL) 306
10.4.5 Thread Status Register Local and Remote (TSRL and TSRR) 307
10.4.6 Thread Switch Control Register (TSCR) 307
10.4.7 Thread Switch Time-Out Register (TTR) 308
10.5 Thread Priority 309
10.5.1 Thread-Priority Combinations 309
10.5.2 Choosing Useful Thread Priorities 310
10.5.3 Examples of Priority Combinations on Instruction Scheduling 312
10.6 Thread Control and Configuration 315
10.6.1 Resuming and Suspending Threads 315
10.6.2 Setting the Instruction-Dispatch Policy: Thread Priority and Temporary Stalling 315
10.6.3 Preventing Starvation: Forward-Progress Monitoring 317
10.6.4 Multithreading Operating-State Switch 318
10.7 Pipeline Events and Instruction Dispatch 318
10.7.1 Instruction-Dispatch Rules 318
10.7.2 Pipeline Events that Stall Instruction Dispatch 319
10.8 Suspending and Resuming Threads 321
10.8.1 Suspending a Thread 321
10.8.2 Resuming a Thread 321
10.8.3 Exception and Interrupt Interactions With a Suspended Thread 323
10.8.4 Thread Targets and Behavior for Interrupts 324

11. Logical Partitions and a Hypervisor 327
11.1 Introduction 327
11.1.1 The Hypervisor and the Operating Systems 328
11.1.2 Partitioning Resources 328
11.1.3 An Example Flowchart 329
11.2 PPE Logical-Partitioning Facilities 331
11.2.1 Enabling Hypervisor State 331
11.2.2 Hypervisor-State Registers 331
11.2.3 Controlling Real Memory 332
11.2.4 Controlling Interrupts and Environment 337
11.3 SPE Logical-Partitioning Facilities 340
11.3.1 Access Privilege 340
11.3.2 Memory-Management Facilities 341
11.3.3 Controlling Interrupts 343
11.3.4 Other SPE Management Facilities 343
11.4 I/O-Address Translation 345
11.4.1 IOC Memory Management Units 345
11.4.2 I/O Segment and Page Tables 345

11.5 Resource Allocation Management 346
11.5.1 Combining Logical Partitions with Resource Allocation 346
11.5.2 Resource Allocation Groups and the Token Manager 346
11.6 Power Management 347
11.6.1 Entering Low-Power States 347
11.6.2 Thread State Suspension and Resumption 347
11.7 Fault Isolation 348
11.8 Code Sample 348
11.8.1 Error Codes and Hypervisor-Call (hcall) Tokens 348
11.8.2 C Functions for PowerPC 64-bit ELF Hypervisor Call 348

12. SPE Context Switching 351
12.1 Introduction 351
12.2 Data Structures 352
12.2.1 Local Store Context Save Area 352
12.2.2 Context Save Area 352
12.3 Overview of SPE Context-Switch Sequence 352
12.3.1 Save SPE Context 354
12.3.2 Restore SPE Context 354
12.4 Implementation Considerations 356
12.4.1 Locking 356
12.4.2 Watchdog Timers 356
12.4.3 Waiting for Events 356
12.4.4 PPE's SPU Channel Access Facility 356
12.4.5 SPE Interrupts 356
12.4.6 Suspending the MFC DMA Queue 357
12.4.7 SPE Context-Save Sequence and Context-Restore Sequence Code 357
12.4.8 SPE Parameter Passing 357
12.4.9 Storage for SPE Context-Save Sequence and Context-Restore Sequence Code 357
12.4.10 Harvesting an SPE 358
12.4.11 Scheduling 358
12.4.12 Light-Weight SPE Context Save 358
12.5 Detailed Steps for SPE Context Switch 359
12.5.1 Context-Save Sequence 359
12.5.2 Context-Restore Sequence 365
12.6 Considerations for Hypervisors 373

13. Time Base and Decrementers 375
13.1 Introduction 375
13.2 Time Base Facility 375
13.2.1 Clock Domains 375
13.2.2 Time-Base Registers 376
13.2.3 Time-Base Frequency 377
13.2.4 Time-Base Sync Mode Controls 378
13.2.5 Reading and Writing the TB Register 382
13.2.6 Computing Time-of-Day 383
13.3 Decrementers 383
13.3.1 PPE Decrementers 383
13.3.2 SPE Decrementers 384

13.3.3 Using an SPU Decrementer to Monitor SPU Code Performance 385

14. Objects, Executables, and SPE Loading 391

14.1 Introduction 391
14.2 ELF Overview and Extensions 392
14.2.1 Overview 392
14.2.2 SPE-ELF Extensions 393
14.3 Runtime Initializations and Requirements 395
14.3.1 PPE Initial Machine State 395
14.3.2 SPE Initial Machine State for Linux 399
14.4 Linker Requirements 401
14.4.1 SPE Linker Requirements 401
14.4.2 PPE Linker Requirements 402
14.5 The CESOF Format 402
14.5.1 CESOF Overview 403
14.5.2 CESOF Use Convention of ELF 403
14.5.3 Embedding an SPE-ELF Executable in a PPE-ELF Object: The `.spu.elf` Section 404
14.5.4 The `spe_program_handle` Data Structure 405
14.5.5 The TOE: Accessing Symbol Values Defined in EA Space 407
14.5.6 Future Software Tool Chain Enhancements for CESOF 411
14.6 SPE Runtime Loader 412
14.6.1 Runtime Loader Overview 412
14.6.2 SPE Runtime Loader Requirements 413
14.6.3 Example SPE Runtime Loader Framework Definition 414
14.7 SPE Execution Environment 421
14.7.1 Signal Types for the SPE Stop-and-Signal Instruction 421

15. Power and Thermal Management 423

15.1 Power Management 423
15.1.1 CBE Slow State 424
15.1.2 PPE Pause (0) State 425
15.1.3 SPU Pause State 426
15.1.4 MFC Pause State 426
15.2 Thermal Management 426
15.2.1 Thermal-Management Operation 427
15.2.2 Configuration-Ring Settings 429
15.2.3 Thermal Registers 429
15.2.4 Thermal Sensor Status Registers 429
15.2.5 Thermal Sensor Interrupt Registers 429
15.2.6 Dynamic Thermal-Management Registers 432

16. Performance Monitoring 437

16.1 How It Works 438
16.2 Events (Signals) 438
16.3 Performance Counters 438
16.4 Trace Array 439

17. SPE Channel and Related MMIO Interface 441

17.1 Introduction 441
17.1.1 An SPE's Use of its Own Channels 441
17.1.2 Access to Channel Functions by the PPE and other SPEs 442
17.1.3 Channel Characteristics 442
17.1.4 Channel Summary 443
17.1.5 Channel Instructions 446
17.1.6 Channel Capacity and Blocking 446
17.2 SPU Event-Management Channels 447
17.3 SPU Signal-Notification Channels 447
17.4 SPU Decrementer 447
17.4.1 SPU Write Decrementer Channel 448
17.4.2 SPU Read Decrementer Channel 448
17.5 MFC Write Multisource Synchronization Request Channel 448
17.6 SPU Read Machine Status Channel 449
17.7 SPU Write State Save-and-Restore Channel 450
17.8 SPU Read State Save-and-Restore Channel 450
17.9 MFC Command Parameter Channels 450
17.9.1 MFC Local Storage Address Channel 452
17.9.2 MFC Effective Address High Channel 453
17.9.3 MFC Effective Address Low or List Address Channel 453
17.9.4 MFC Transfer Size or List Size Channel 454
17.9.5 MFC Command Tag Identification Channel 455
17.9.6 MFC Class ID and MFC Command Opcode Channel 455
17.10 MFC Tag-Group Management Channels 456
17.10.1 MFC Write Tag-Group Query Mask Channel 457
17.10.2 MFC Read Tag-Group Query Mask Channel 457
17.10.3 MFC Write Tag Status Update Request Channel 457
17.10.4 MFC Read Tag-Group Status Channel 458
17.10.5 MFC Read List Stall-and-Notify Tag Status Channel 459
17.10.6 MFC Write List Stall-and-Notify Tag Acknowledgment Channel 460
17.11 MFC Read Atomic Command Status Channel 461
17.12 SPU Mailbox Channels 461

18. SPE Events 463
18.1 Introduction 463
18.2 Events and Event-Management Channels 464
18.2.1 Event Conditions and Bit Definitions for Event-Management Channels 464
18.2.2 Pending Event Register (Internal, SPE-Hidden) 465
18.2.3 SPU Read Event Status 466
18.2.4 SPU Write Event Mask 467
18.2.5 SPU Write Event Acknowledgement 467
18.2.6 SPU Read Event Mask 468
18.3 SPU Interrupt Facility 468
18.4 Interrupt Address Save-and-Restore Channels 469
18.4.1 SPU Read State Save-and-Restore 469
18.4.2 SPU Write State Save-and-Restore 469
18.4.3 Nested Interrupts Using SPU Write State Save-and-Restore 469
18.5 Event-Handling Protocols 470
18.5.1 Synchronous Event Handling Using Polling or Stalling 470

18.5.2 Asynchronous Event Handling Using Interrupts 471
18.5.3 Protecting Critical Sections from Interruption 472
18.6 Event-Specific Handling Guidelines 472
18.6.1 Protocol With Multiple Events Enabled 473
18.6.2 Procedure for Handling the Multisource Synchronization Event 474
18.6.3 Procedure for Handling the Privileged Attention Event 476
18.6.4 Procedure for Handling the Lock-Line Reservation Lost Event 476
18.6.5 Procedure for Handling the Signal-Notification 1 Available Event 477
18.6.6 Procedure for Handling the Signal-Notification 2 Available Event 478
18.6.7 Procedure for Handling the SPU Write Outbound Mailbox Available Event 479
18.6.8 Procedure for Handling the SPU Write Outbound Interrupt Mailbox Available Event 480
18.6.9 Procedure for Handling the SPU Decrementer Event 480
18.6.10 Procedure for Handling the SPU Read Inbound Mailbox Available Event 482
18.6.11 Procedure for Handling the MFC SPU Command Queue Available Event 483
18.6.12 Procedure for Handling the DMA List Command Stall-and-Notify Event 484
18.6.13 Procedure for Handling the Tag-Group Status Update Event 486
18.7 Developing a Basic Interrupt Handler 486
18.7.1 Basic Interrupt Protocol Features and Design 487
18.7.2 FLIH Design 487
18.7.3 SLIH Design and Registering SLIH Functions 489
18.7.4 Example Application Code 491
18.8 Nested Interrupt Handling 493
18.8.1 Nested Handler Design 493
18.8.2 FLIH Design for Nested Interrupts 493
18.9 Using a Dedicated Interrupt Stack 495
18.10 Sample Applications 497
18.10.1 SPU Decrementer Event 497
18.10.2 Tag-Group Status Update Event 499
18.10.3 DMA List Command Stall-and-Notify Event 499
18.10.4 MFC SPU Command Queue Available Event 502
18.10.5 SPU Read Inbound Mailbox Available Event 502
18.10.6 SPU Signal-Notification Available Event 502
18.10.7 Lock-Line Reservation Lost Event 503
18.10.8 Privileged Attention Event 503

19. DMA Transfers and Inter-Processor Communication 505
19.1 Introduction 505
19.2 MFC Commands 506
19.2.1 DMA Commands 508
19.2.2 DMA List Commands 509
19.2.3 Synchronization Commands 510
19.2.4 Command Modifiers 510
19.2.5 Tag Groups 511
19.2.6 MFC Command Issue 512
19.2.7 Replacement Class ID and Transfer Class ID 513
19.2.8 DMA-Command Completion 514
19.3 PPE-Initiated DMA Transfers 514
19.3.1 MFC Command Issue 515
19.3.2 MFC Command-Queue Control Registers 516

19.3.3 DMA-Command Issue Status and Errors 517
19.4 SPE-Initiated DMA Transfers 521
19.4.1 MFC Command Issue 521
19.4.2 MFC Command-Queue Monitoring Channels 523
19.4.3 DMA Command Issue Status and Errors 523
19.4.4 DMA List Command Example 527
19.5 Performance Guidelines for MFC Commands 530
19.6 Mailboxes 531
19.6.1 Reading and Writing Mailboxes 532
19.6.2 Mailbox Blocking 532
19.6.3 Dealing with Anticipated Messages 532
19.6.4 Uses of Mailboxes 533
19.6.5 SPU Outbound Mailboxes 533
19.6.6 SPU Inbound Mailbox 538
19.7 Signal Notification 542
19.7.1 SPU Signalling Channels 543
19.7.2 Uses of Signaling 544
19.7.3 Mode Configuration 544
19.7.4 SPU Signal Notification 1 Channel 544
19.7.5 SPU Signal Notification 2 Channel 544
19.7.6 Sending Signals 545
19.7.7 Receiving Signals 548
19.7.8 Differences Between Mailboxes and Signal Notification 550

20. Shared-Storage Synchronization 551
20.1 Shared-Storage Ordering 551
20.1.1 Storage Model 551
20.1.2 PPE Ordering Instructions 554
20.1.3 SPU Ordering Instructions 558
20.1.4 MFC Ordering Mechanisms 562
20.1.5 MFC Multisource Synchronization Facility 567
20.1.6 Scenarios for Using Ordering Mechanisms 574
20.2 PPE Atomic Synchronization 575
20.2.1 Atomic Synchronization Instructions 575
20.2.2 PPE Synchronization Primitives 577
20.2.3 SPE Synchronization Primitives 580
20.3 SPE Atomic Synchronization 587
20.3.1 MFC Commands for Atomic Updates 587
20.3.2 The MFC Read Atomic Command Status Channel 589
20.3.3 Avoiding Livelocks 590
20.3.4 Synchronization Primitives 591

21. Parallel Programming 599
21.1 Challenges 599
21.2 Patterns of Parallel Programming 599
21.2.1 Terminology 600
21.2.2 Finding Parallelism 601
21.2.3 Strategies for Parallel Programming 602
21.3 Steps for Parallelizing a Program 604

21.3.1 Step 1: Understand the Problem 604
21.3.2 Step 2: Choose Programming Tools and Technology 604
21.3.3 Step 3: Develop High-Level Parallelization Strategy 605
21.3.4 Step 4: Develop Low-Level Parallelization Strategy 605
21.3.5 Step 5: Design Data Structures for Efficient Processing 605
21.3.6 Step 6: Iterate and Refine 606
21.3.7 Step 7: Fine-Tune 606
21.4 Levels of Parallelism in the CBE Processor 607
21.4.1 SIMD Parallelization 608
21.4.2 Superscalar Parallelization 608
21.4.3 Hardware Multithreading 608
21.4.4 Multiple Execution Units 608
21.4.5 Multiple CBE Processors 609
21.5 Tools for Parallelization 610
21.5.1 Language Extensions: Intrinsics and Directives 610
21.5.2 Compiler Support for Single Shared-Memory Abstraction 611
21.5.3 OpenMP Directives 611
21.5.4 Compiler-Controlled Software Cache 613
21.5.5 Compiler and Runtime Support for Code Partitioning 616
21.5.6 Thread Library 617

22. SIMD Programming 619
22.1 SIMD Basics 619
22.1.1 Converting Scalar Data to SIMD Data 620
22.1.2 Approaching SIMD Coding Methodically 624
22.1.3 Coding for Effective Auto-SIMDization 635
22.2 Auto-SIMDizing Compilers 637
22.2.1 Motivation and Challenges 638
22.2.2 Examples of Invalid and Valid SIMDization 640
22.3 SIMDization Framework for a Compiler 644
22.3.1 Phase 1: Basic-Block Aggregation 646
22.3.2 Phase 2: Short-Loop Aggregation 646
22.3.3 Phase 3: Loop-Level Aggregation 647
22.3.4 Phase 4: Alignment Devirtualization 648
22.3.5 Phase 5: Length Devirtualization 653
22.3.6 Phase 6: SIMD Code Generation and Instruction Scheduling 654
22.3.7 SIMDization Example: Multiple Sources of SIMD Parallelism 655
22.3.8 SIMDization Example: Multiple Data Lengths 658
22.3.9 Vector Operations and Mixed-Mode SIMDization 663
22.4 Other Compiler Optimizations 664
22.4.1 OpenMP 664
22.4.2 Subword Data Types 664
22.4.3 Backend Scheduling for SPEs 665
22.4.4 Interacting with Typical Optimizations 666

23. Vector/SIMD Multimedia Extension and SPU Programming 669
23.1 Architectural Differences 669
23.1.1 Registers 670
23.1.2 Data Types 671

23.1.3 Instruction-Set Differences 672
23.2 Porting SIMD Code from the PPE to the SPEs 673
23.2.1 Code-Mapping Considerations 674
23.2.2 Simple Macro Translation 675
23.2.3 Full Functional Mapping 678
23.2.4 Code-Portability Typedefs 679
23.2.5 Compiler-Target Definition 679

24. SPE Programming Tips 681
24.1 DMA Transfers 681
24.1.1 Initiating DMA Transfers from SPEs 682
24.1.2 Overlapping DMA Transfers and Computation 682
24.1.3 DMA Transfers and LS Accesses 687
24.2 SPU Pipelines and Dual-Issue Rules 687
24.3 Eliminating and Predicting Branches 689
24.3.1 Function-Inlining and Loop-Unrolling 689
24.3.2 Predication Using Select-Bits Instruction 689
24.3.3 Branch Hints 691
24.3.4 Program-Based Branch Prediction 694
24.3.5 Profile or Linguistic Branch-Prediction 696
24.3.6 Software Branch-Target Address Cache 697
24.3.7 Using Control Flow to Record Branch History 697
24.4 Loop Unrolling and Pipelining 698
24.5 Offset Pointers 701
24.6 Transformations and Table-Lookups 701
24.6.1 The Shuffle-Bytes Instruction 701
24.6.2 Fast SIMD 8-Bit Table Lookups 702
24.7 Integer Multiplies 705
24.8 Scalar Code 705
24.8.1 Scalar Loads and Stores 705
24.8.2 Promoting Scalar Data Types to Vector Data Types 706
24.9 Unaligned Loads 707

Appendix A. PPE Instruction Set and Intrinsics 711
A.1 PowerPC Instruction Set 711
A.1.1 Data Types 711
A.1.2 Instructions 711
A.1.3 Microcoded Instructions 723
A.2 PowerPC Extensions in the PPE 729
A.2.1 New PowerPC Instructions 729
A.2.2 Implementation-Dependent Interpretation of PowerPC Instructions 733
A.2.3 Optional PowerPC Instructions Implemented 736
A.2.4 PowerPC Instructions Not Implemented 736
A.2.5 Endian Support 737
A.3 Vector/SIMD Multimedia Extension Instructions 737
A.3.1 Data Types 737
A.3.2 Instructions 738
A.3.3 Graphics Rounding Mode 742

A.4 C/C++ Language Extensions (Intrinsics) for Vector/SIMD Multimedia Extensions 744
A.4.1 Vector Data Types 744
A.4.2 Vector Literals 745
A.4.3 Intrinsics 746
A.5 Issue Rules 750
A.6 Pipeline Stages 752
A.6.1 Instruction-Unit Pipeline 752
A.6.2 Vector/Scalar Unit Issue Queue 754
A.6.3 Stall and Flush Points 755
A.7 Compiler Optimizations 757
A.7.1 Instruction Arrangement 757
A.7.2 Avoiding Slow Instructions and Processor Modes 757
A.7.3 Avoiding Dependency Stalls and Flushes 758
A.7.4 General Recommendations 760

Appendix B. SPU Instruction Set and Intrinsics 761
B.1 SPU Instruction Set 761
B.1.1 Data Types 761
B.1.2 Instructions 761
B.1.3 Fetch and Issue Rules 768
B.1.4 Inline Prefetch and Instruction Runout 771
B.2 C/C++ Language Extensions (Intrinsics) for SPU Instructions 772
B.2.1 Vector Data Types 772
B.2.2 Vector Literals 774
B.2.3 Intrinsics 776
B.2.4 Inline Assembly 779
B.2.5 Compiler Directives 779

Appendix C. Performance Monitor Signals 781
C.1 Selecting Performance Monitor Signals on the CBE Debug Bus 781
C.1.1 An Example of Setting up the Performance Monitor in PPSS L2 Mode A 783
C.2 PowerPC Processing Unit (PPU) Signal Selection 785
C.2.1 PPU Instruction Unit (NClk) 785
C.2.2 PPU Execution Unit (NClk) 786
C.3 PowerPC Storage Subsystem (PPSS) Signal Selection 787
C.3.1 PPSS Bus Interface Unit (NClk/2) 787
C.3.2 PPSS L2 Cache Controller - Group 1 (NClk/2) 788
C.3.3 PPSS L2 Cache Controller - Group 2 (NClk/2) 789
C.3.4 PPSS L2 Cache Controller - Group 3 (NClk/2) 790
C.3.5 PPSS Non-Cacheable Unit (NClk/2) 791
C.4 Synergistic Processor Unit (SPU) Signal Selection 792
C.4.1 SPU (NClk) 792
C.4.2 SPU Trigger (NClk) 794
C.4.3 SPU Event (NClk) 795
C.5 Memory Flow Control (MFC) Signal Selection 796
C.5.1 MFC Atomic Unit (NClk/2) 796
C.5.2 MFC Direct Memory Access Controller (NClk/2) 797
C.5.3 MFC Synergistic Bus Interface (NClk/2) 798
C.5.4 MFC Synergistic Memory Management (NClk/2) 799

C.6 Element Interconnect Bus (EIB) Signal Selection 800
C.6.1 EIB Address Concentrator 0 (NClk/2) 800
C.6.2 EIB Address Concentrator 1 (NClk/2) 801
C.6.3 EIB Data Ring Arbitrator - Group 1 (NClk/2) 802
C.6.4 EIB Data Ring Arbiter - Group 2 (NClk/2) 803
C.6.5 EIB Token Manager (NClk/2) 804
C.7 Memory Interface Controller (MIC) Signal Selection 810
C.7.1 MIC Group 1 (NClk/2) 810
C.7.2 MIC Group 2 (NClk/2) 811
C.7.3 MIC Group 3 (NClk/2) 812
C.8 Broadband Engine Interface (BEI) Unit 813
C.8.1 BIF Controller - IOIF0 Word 0 (NClk/2) 814
C.8.2 BIF Controller - IOIF1 Word 0 (NClk/2) 815
C.8.3 BIF Controller - IOIF0 Word 2 (NClk/2) 816
C.8.4 BIF Controller - IOIF1 Word 2 (NClk/2) 817
C.8.5 I/O Controller Word 0 - Group 1 (NClk/2) 818
C.8.6 I/O Controller Word 2 - Group 2 (NClk/2) 819
C.8.7 I/O Controller - Group 3 (NClk/2) 820
C.8.8 I/O Controller Word 0 - Group 4 (NClk/2) 821

Glossary 823

Index 859

Revision Log 875

List of Figures

Figure 1-1. Cell Broadband Engine Overview 33
Figure 1-2. Storage and Domains and Interfaces 40
Figure 1-3. Big-Endian Byte and Bit Ordering 42
Figure 2-1. PPE Block Diagram 45
Figure 2-2. PPE Functional Units 46
Figure 2-3. Concurrent Execution of Fixed-Point, Floating-Point, and Vector Extension Units 47
Figure 2-4. PPE User Register Set 49
Figure 2-5. Four Concurrent Add Operations 56
Figure 2-6. Byte-Shuffle (Permute) Operation 56
Figure 3-1. SPE Block Diagram 63
Figure 3-2. SPU Functional Units 64
Figure 3-3. LS Access Methods 65
Figure 3-4. SPE Problem-State (User) Register Set 67
Figure 3-5. MFC Block Diagram 71
Figure 3-6. Register Layout of Data Types and Preferred Scalar Slot 75
Figure 4-1. PPE Address-Translation Overview 82
Figure 4-2. SLB Entry Format 85
Figure 4-3. Mapping a Virtual Address to a Real Address 93
Figure 4-4. Real-Mode I and G Bit Settings 101
Figure 4-5. SPE Effective- to Virtual-Address Mapping and TLB-Index Selection 105
Figure 4-6. SMM Virtual- to Real-Address Mapping 108
Figure 4-7. Pseudo-LRU Binary Tree for SPE TLB 110
Figure 4-8. MFC TLB VPN Tag 114
Figure 4-9. Real-Mode Storage Boundary 117
Figure 5-1. CBE Memory Map 122
Figure 6-1. PPE Cache 135
Figure 6-2. Pseudo-LRU Binary Tree for L2-Cache LRU Binary Tree with Locked Ways A, C, and H 144
Figure 6-3. Generation of RclassID from the Address Range Registers for Each PPE Thread 157
Figure 7-1. BEI and its Two I/O Interfaces 161
Figure 7-2. Single and Dual CBE System Configurations 163
Figure 7-3. Quad CBE System Configuration 164
Figure 7-4. Use of I/O Addresses by an I/O Device 165
Figure 7-5. I/O-Address Translation Overview 178
Figure 7-6. I/O-Address Translation Using Cache 179
Figure 7-7. IOPT Cache Hash and IOPT Cache Directory Tag 186
Figure 8-1. Resource Allocation Overview 203
Figure 8-2. Managed Resources and Requesters 204
Figure 8-3. Token Manager 213

Figure 8-4. Banks, Rows, and Columns 215
Figure 8-5. Memory-Token Example for TKM_CR[MT] = ‘10’ 220
Figure 8-6. Threshold and Queue Levels 226
Figure 9-1. Organization of Interrupt Handling 240
Figure 10-1. PPE Multithreading Pipeline Flow and Resources 300
Figure 10-2. Thread Priorities and Instruction Dispatch: Low Priority with Low Priority 312
Figure 10-3. Example 2 of Thread Priorities and Instruction Dispatch 313
Figure 10-4. Example 3 of Thread Priorities and Instruction Dispatch 313
Figure 10-5. Example 4 of Thread Priorities and Instruction Dispatch 314
Figure 10-6. Example 5 of Thread Priorities and Instruction Dispatch 314
Figure 11-1. Overview of Logical Partitions 327
Figure 11-2. Flowchart of overview of Hypervisor Management of Logical Partitions 330
Figure 11-3. Operating-System and Hypervisor Views of Physical Memory 334
Figure 12-1. SPE Context-Switch Sequence 353
Figure 13-1. Time Base (TB) Register 376
Figure 13-2. Internal Time-Base Sync Mode Initialization Sequence 379
Figure 13-3. External Time-Base Sync Mode Initialization Sequence 382
Figure 14-1. Object-File Format 392
Figure 14-2. PPE 64-Bit Initial Stack Frame 397
Figure 14-3. PPE 64-Bit Standard Stack Frame 398
Figure 14-4. SPE Initial Stack Frame 400
Figure 14-5. SPE Standard Application Function Stack Frame 401
Figure 14-6. Linking SPE-ELF Sections into Loadable Segments in an Executable 404
Figure 14-7. CESOF Object Layout 410
Figure 14-8. PPE OS Loads CESOF Object and spu_ld.so into EA Space 417
Figure 14-9. PPE OS Loads `spu_ld.so` and Parameters into SPE LS 418
Figure 14-10. `spu_ld.so` Loads SPE Code and TOE Shadow 420
Figure 15-1. Digital Thermal-Sensor Throttling 428
Figure 17-1. SPE Access to Its Own Channels 441
Figure 17-2. Sequences for Issuing MFC Commands 452
Figure 18-1. SPE Search of Ordered Binary Tree 501
Figure 19-1. DMA List Element 528
Figure 20-1. Barriers and Fences 565
Figure 20-2. PPE Multisource Synchronization Flowchart 569
Figure 20-3. SPE Multisource Synchronization Flowchart (Non-Event-Based) 571
Figure 20-4. SPE Multisource Synchronization Flowchart (Event-Based) 572
Figure 21-1. Software-Cache Lookup 615
Figure 22-1. A Vector With Four Elements 619
Figure 22-2. Array-of-Structures (AOS) Data Organization 621

Figure 22-3. Structure-of-Arrays (SOA) Data Organization 621
Figure 22-4. Triangle Subdivision 622
Figure 22-5. Traditional Sequential Implementation of a Loop 625
Figure 22-6. Simple, Incorrect SIMD Implementation of a Loop 626
Figure 22-7. Simple, Correct SIMD Implementation of a Loop 628
Figure 22-8. Optimized, Correct SIMD Implementation of a Loop 629
Figure 22-9. Implementing Stream-Shift Operations for a Loop 631
Figure 22-10. SIMDization Prolog 632
Figure 22-11. SIMDization Epilog 633
Figure 22-12. Invalid SIMDization on Hardware with Alignment Constraints 641
Figure 22-13. An Example of a Valid SIMDization on SIMD Unit with Alignment Constraints 643
Figure 22-14. SIMDization Framework Based on Virtual Vectors 645
Figure 22-15. Eager-Shift Policy 650
Figure 22-16. Lazy-Shift Policy 651
Figure 22-17. Dominant-Shift Policy 652
Figure 22-18. Implementing Partial Vector Store 653
Figure 22-19. Implementing Stream Shift 655
Figure 22-20. SIMDization Example with Mixed Stride-One and Adjacent Accesses 657
Figure 22-21. Length Conversion After Loop-Level SIMDization 660
Figure 22-22. Length Conversion After Devirtualization 662
Figure 24-1. Serial Computation and Transfer 682
Figure 24-2. DMA Transfers Using a Double-Buffering Method 683
Figure 24-3. Parallel Computation and Transfer 683
Figure 24-4. Shared I/O Buffers with Fenced get Command 686
Figure 24-5. Branch Stall Window 693
Figure 24-6. Basic Loop 699
Figure 24-7. Software-Pipelined Loop 700
Figure 24-8. Shuffle Bytes Instruction 702
Figure 24-9. 128-Entry Table Lookup Example 704
Figure A-1. Dual-Issue Combinations 751
Figure A-2. BRU, FXU, and LSU Pipeline for PPE 753
Figure A-3. VSU (FPU and VXU) Pipeline for PPE 754
Figure 24-10. Element Ordering for Vector Types 774

List of Tables

Table 1-1. Definition of Threads and Tasks 42
Table 2-1. PowerPC Data Types 51
Table 2-2. PowerPC Instruction Summary 53
Table 2-3. Vector/SIMD Multimedia Extension Instruction Summary 58
Table 3-1. LS-Access Arbitration Priority and Transfer Size 66
Table 3-2. SPE Floating-Point Support 69
Table 3-3. SPU Instruction Summary 75
Table 4-1. PPE Memory Management Unit Features 81
Table 4-2. PTE Fields 87
Table 4-3. Decoding for Large Page Sizes 90
Table 4-4. Summary of Implemented WIMG Settings 91
Table 4-5. TLB Compare-Tag Width 94
Table 4-6. TLB Software Management 96
Table 4-7. Supported Invalidation Selector (IS) Values in the tlbiel Instruction 98
Table 4-8. Summary of Real Addressing Modes 100
Table 4-9. Summary of Real-Mode Attributes 100
Table 4-10. Real-Mode Storage Control Values 100
Table 4-11. SPE Synergistic Memory Management Unit Features 102
Table 4-12. SLB and TLB Compare Tags 109
Table 4-13. SMM_HID Page-Size Decoding 112
Table 4-14. MFC_TLB_Invalidate_Entry Index Selection 114
Table 4-15. Translation, MMIO, and Address Match-Generated Exceptions 117
Table 5-1. Memory-Base Registers Loaded by Configuration Ring 123
Table 5-2. CBE Memory Map 124
Table 5-3. PPE Privileged Memory-Mapped Register Groups 126
Table 5-4. Predefined Real-Address Locations 127
Table 5-5. SPE Problem-State Memory-Mapped Register Groups 129
Table 5-6. SPE Privilege-State-2 Memory-Mapped Register Groups 129
Table 5-7. SPE Privilege-State-1 Memory-Mapped Register Groups 129
Table 6-1. Atomic-Unit Commands 153
Table 6-2. Address Range Registers (One per PPE Thread) 156
Table 7-1. IOIF Mask and Size of Address Range 175
Table 7-2. Memory Coherence Requirements for I/O Accesses 190
Table 7-3. Ordering of Inbound I/O Accesses from Same or Different I/O Addresses 192
Table 7-4. Ordering of Inbound I/O Accesses from Different I/O Addresses 192
Table 8-1. Tokens Needed for EIB or IOIF Operations in Single-CBE Systems 207
Table 8-2. Memory Tokens Available Based on TKM_CR[MT] 217
Table 8-3. OR and AND Values for All RAG Priorities 224

Table 8-4. Token Allocation Actions Per Resource Queue Level 227
Table 8-5. Configuration Bits in the PPE, SPEs, IOC, and MIC 229
Table 8-6. SPE Address-Range Registers 230
Table 8-7. MC_COMP_EN Setting 230
Table 8-8. IOIF1_COMP_EN and BE_MMIO_Base Settings 231
Table 9-1. PPE Interrupts 239
Table 9-2. PPE Interrupt Register Summary 241
Table 9-3. Interrupt Vector and Exception Conditions 244
Table 9-4. Registers Altered by a System Reset Interrupt 246
Table 9-5. Registers Altered by a Machine Check Interrupt 248
Table 9-6. Registers Altered by a Data Storage Interrupt 249
Table 9-7. Registers Altered by a Data Segment Interrupt 250
Table 9-8. Registers Altered by an Instruction Storage Interrupt 251
Table 9-9. Registers Altered by an Instruction Segment Interrupt 251
Table 9-10. Registers Altered by an Alignment Interrupt 252
Table 9-11. Registers Altered by a Program Interrupt 254
Table 9-12. Registers by a Floating-Point Unavailable Interrupt 254
Table 9-13. Registers Altered by a Decrementer Interrupt 254
Table 9-14. Registers Altered by a Hypervisor Decrementer Interrupt 255
Table 9-15. Registers Altered by a System Call Interrupt 255
Table 9-16. Registers Altered by a Trace Interrupt 256
Table 9-17. Registers Altered by a VXU Unavailable Interrupt 257
Table 9-18. Registers Altered by a System Error 257
Table 9-19. Registers Altered by a Maintenance Interrupt 260
Table 9-20. Registers Altered by a Thermal Management Interrupt 261
Table 9-21. Interrupt-Destination Unit ID 263
Table 9-22. IOC Interrupt Register Summary 264
Table 9-23. IIC_IPP0 and IIC_IPP1 Interrupt Pending Port Bit Fields 266
Table 9-24. Values for Interrupt Source 266
Table 9-25. IIC_IGP0 and IIC_IGP1 Interrupt Generation Port Bit Field 268
Table 9-26. IIC_IRR Routing Fields 270
Table 9-27. Registers Altered by an External Interrupt 276
Table 9-28. SPU and MFC External Interrupt Definitions 277
Table 9-29. SPE Interrupt Register Summary 278
Table 9-30. INT_Mask_class0 Bit Fields 280
Table 9-31. SPU_ERR_Mask Bit Fields 281
Table 9-32. INT_Mask_class1 Bit Fields 281
Table 9-33. INT_Mask_class2 Bit Fields 281
Table 9-34. INT_Route Bit Fields 282

Table 9-35. CBE Unit Values 282
Table 9-36. Exceptions Generated by Translation, MMIO, and Address Matches 282
Table 9-37. Format for Interrupt Packet 286
Table 9-38. PPE-Thread Target for Interrupts 287
Table 9-39. Priority Order of Interrupt Conditions 288
Table 9-40. Machine State Register (MSR) Bit-Settings Due to Interrupts 290
Table 10-1. PPE Multithreading versus Multi-Core Implementations 295
Table 10-2. Privilege States 301
Table 10-3. Effect of Setting the `CTRL[TE0]` and `CTRL[TE1]` Bits 306
Table 10-4. Relative Thread Priorities and Instruction-Dispatch Policies 310
Table 10-5. Three Useful Thread Priority Combinations 311
Table 10-6. Privilege States and Allowed Thread-Priority Changes 316
Table 10-7. **nop** Instructions to Set Thread Priority and Stall Instruction Dispatch 316
Table 10-8. Thread Resume Reason Code 322
Table 10-9. Interrupt Entry, Masking, and Multithreading Behavior 324
Table 11-1. Summary of Real Mode Limit Select (RMLS) Values 335
Table 13-1. TBR[Timebase_setting] for Reference Divider (RefDiv) Setting 379
Table 14-1. SPE-ELF Header Fields 393
Table 14-2. SPE-ELF Special Sections 394
Table 14-3. SPE-ELF Environment Note 394
Table 14-4. The `spu_env` Structure 395
Table 14-5. SPE-ELF Name Note 395
Table 14-6. PPE Program Initial Parameters 396
Table 14-7. PPE Initial Register State 396
Table 14-8. SPE Program Initial Parameters 399
Table 14-9. SPE Register Initial Values 400
Table 14-10. SPE Loader Per-Loader Parameters 415
Table 14-11. SPE Loader Per-Binary Parameters 416
Table 14-12. LS Locations and Sizes of SPE Loader and Parameters 419
Table 14-13. SPE-ELF Reserved Signal Type Values 422
Table 15-1. Power-Management States 423
Table 15-2. Register Updates for CBE Slow State 425
Table 15-3. Digital Temperature-Sensor Encoding 427
Table 17-1. SPE Channel Types and Characteristics 442
Table 17-2. SPE Channels and Associated MMIO Registers 443
Table 17-3. SPE Channel Instructions 446
Table 17-4. MFC Command Parameter and Command Opcode Channels 450
Table 18-1. Bit Assignments for Event-Management Channels and Internal Register 464
Table 18-2. Bit Definitions for Event-Management Channels and Internal Register 464

Table 18-3. Indirect-Branch Instructions 471
Table 18-4. SPE Event Handling Protocol Summaries 473
Table 19-1. SPE DMA and inter-Processor Communication Mechanisms 505
Table 19-2. Comparison of Mailboxes and Signals 506
Table 19-3. MFC Command Queues 507
Table 19-4. MFC DMA Put Commands 508
Table 19-5. MFC DMA Get Commands 509
Table 19-6. MFC Synchronization Commands 510
Table 19-7. MFC Command-Modifier Suffixes 511
Table 19-8. MFC Issue Slot Command Assignments 513
Table 19-9. MFC Command-Parameter Registers for PPE-Initiated DMA Transfers 515
Table 19-10. MFC_CMDStatus Return Values 516
Table 19-11. MFC Command-Queue MMIO Registers for PPE-Initiated Commands 517
Table 19-12. MFC Command-Parameter Channels for SPE-Initiated DMA Transfers 521
Table 19-13. MFC Tag-Status Channels for Monitoring SPE-Initiated Commands 523
Table 19-14. MFC Channels for Event Monitoring and Management 523
Table 19-15. Mailbox Channels and MMIO Registers 531
Table 19-16. Functions of Mailbox Channels 531
Table 19-17. Functions of Mailbox MMIO Registers 531
Table 19-18. Fields of Mailbox Status Register (SPU_Mbox_Stat) 539
Table 19-19. Signal-Notification Channels and MMIO Registers 543
Table 19-20. Functions of Signal-Notification Channels 543
Table 19-21. Functions of Signal-Notification MMIO Registers 543
Table 20-1. Effects of Synchronization on Address and Communication Domains 553
Table 20-2. Storage-Barrier Function Summary 557
Table 20-3. Storage-Barrier Ordering of Accesses to System Memory 557
Table 20-4. Storage-Barrier Ordering of Accesses to Device Memory 558
Table 20-5. SPU Synchronization Instructions 561
Table 20-6. Synchronization Instructions for Accesses to an LS 561
Table 20-7. Tag-Specific Ordering Commands 564
Table 20-8. MFC Multisource Synchronization Facility Channel and MMIO Register 568
Table 20-9. MFC Commands for Atomic Updates 588
Table 20-10. MFC Read Atomic Command Status Channel Contents 590
Table 21-1. Parallel-Programming Terminology 600
Table 23-1. PPE and SPE SIMD-Support Comparison 669
Table 23-2. Vector/SIMD Multimedia Extension and SPU Vector Data Types 671
Table 23-3. Additional Vector/SIMD Multimedia Extension Single-Token Data Types 676
Table 23-4. SPU Intrinsics with One-to-One Vector/SIMD Multimedia Extension Mapping 677
Table 23-5. Vector/SIMD Multimedia Extension Intrinsics with One-to-One SPU Mapping 677

Table 23-6. Unique SPU-to-Vector/SIMD Multimedia Extension Data-Type Mappings 678
Table 24-1. Pipeline 0 Instructions and Latencies 688
Table 24-2. Pipeline 1 Instructions and Latencies 688
Table 24-3. Hint-for Branch Instructions 691
Table 24-4. Heuristics for Static Branch Prediction 694
Table 24-5. Loop Unrolling the `xformlight` Workload 699
Table 24-6. Software Pipelining the Unrolled `xformlight` Workload 700
Table 24-7. Performance Comparison 704
Table 24-8. RC4 Scalar Improvements 706
Table 24-9. Intrinsics for Changing Scalar and Vector Data Types 707
Table 24-10. Preferred Scalar Slot 707
Table A-1. PowerPC Instructions, by Execution Unit 712
Table A-2. Storage Alignment for PowerPC Instructions 722
Table A-3. Unconditionally Microcoded Instructions (Except Loads and Stores) 724
Table A-4. Unconditionally Microcoded Loads and Stores 726
Table A-5. Conditionally Microcoded Instructions 728
Table A-6. Summary of Real Mode Limit Select (RMLS) Values 733
Table A-7. Summary of Supported IS Values in TLBIEL 734
Table A-8. Vector/SIMD Multimedia Extension Instructions 738
Table A-9. Storage Alignment for Vector/SIMD Multimedia Extension Instructions 742
Table A-10. Vector/SIMD Multimedia Extension Data Types 744
Table A-11. AltiVec Vector-Literal Format 745
Table A-12. Curly-Brace Vector-Literal Format 746
Table A-13. Vector/SIMD Multimedia Extension Intrinsics 746
Table B-1. SPU Instructions 762
Table B-2. Instruction Issue Example 769
Table B-3. Example of Instruction Issue without NOP and LNOP 769
Table B-4. Example of Instruction Issue Using NOP and LNOP 770
Table B-5. Execution Pipelines and Result Latency 771
Table B-6. Vector Data Types 772
Table B-7. Programmer Typedefs for Vector Data Types 773
Table B-8. Data Type Alignments 773
Table B-9. Vector-Literal Format 774
Table B-10. Alternate Vector-Literal Format 775
Table B-11. SPU Intrinsics 776
Table C-1. count_cycles Field of the PMx_control Register 782
Table C-2. Lanes Used by Different Islands on the Debug Bus 782
Table C-3. Register Settings to Enable Pass-Through of Debug Bus Signals Through Islands 783

Preface

This handbook is a general programming reference for the Cell Broadband Engine™ (CBE) processor, with extended content targeted for programmers who are interested in developing user applications, libraries, drivers, middleware, compilers, or operating systems for the CBE processor. The chapters describe all of the facilities unique to the CBE processor that are needed in order to develop such programs.

In general, the chapters contain the following topics:

- General hardware and programming-environment overview: Chapters 1 through 3.
- Additional hardware overview plus privilege-state (supervisor) programming: Chapters 4 through 16. A few user-programming topics, such as decrementers, are also located in these chapters.
- Problem-state (user) programming: Chapters 17 through 24, and appendices.

The handbook assumes that the reader is an experienced C/C++ programmer and familiar with basic concepts of single-instruction, multiple-data (SIMD) vector instruction sets, such as the PowerPC® Architecture Vector/SIMD Multimedia Extensions, AltiVec™, Intel® MMX™, SSE, 3DNOW!, x86-64, or VIS instruction sets. The handbook is system-independent; it makes no assumptions about development-tool or operating-system environments.

The handbook should provide adequate detail for general software development. Additional specific implementation details may be available under non-disclosure agreement from either Sony, Toshiba, or IBM. System-specific details should be available from the specific system supplier.

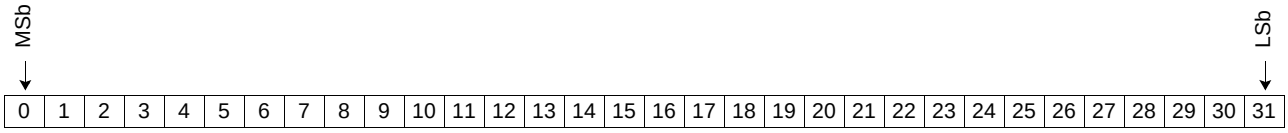
Related Publications

Title	Version	Date
Cell Broadband Engine Architecture	1.0	July 8, 2005
PowerPC User Instruction Set Architecture, Book I	2.02	January 28, 2005
PowerPC Virtual Environment Architecture, Book II	2.02	January 28, 2005
PowerPC Operating Environment Architecture, Book III	2.02	January 28, 2005
PowerPC Microprocessor Family: The Programming Environments for 64- and 32-Bit Microprocessors	2.0	June 10, 2003
PowerPC Microprocessor Family: Vector/SIMD Multimedia Extension Technolgy Programming Environments Manual	2.06c	September 30, 2005
Synergistic Processor Unit Instruction Set Architecture	1.1	January 30, 2006
SPU C/C++ Language Extensions	2.1	October 20, 2005
SPU Application Binary Interface Specification	1.4	October 20, 2005
SPU Assembly Language Specification	1.3	October 20, 2005
Cell Broadband Engine Registers	1.0	November 9, 2005
PowerPC Compiler Writer's Guide	1.0	1996
Cell Broadband Engine Hardware Initialization Guide		2006

Title	Version	Date
Linux™ Standard Base Core Specification for PPC 3.0 (http://www.linuxbase.org/spec).	3.0	2004
64-bit PowerPC ELF Application Binary Interface Supplement 1.7.1.	1.7.1	July 21, 2004
Cell Broadband Engine Linux Reference Implementation Application Binary Interface Specification.	1.0	November 9, 2005
AltiVec Technology Programming Interface Manual	0	June, 1999
IBM Research Hypervisor (rHype) (This is available in source form with GPL license at: http://www.research.ibm.com/hypervisor/)	1.3	March 11, 2005
Tool Interface Standard (TIS) Executable and Linking Format (ELF) specification. TIS Committee.	1.2	May, 1995
SYSTEM V APPLICATION BINARY INTERFACE PowerPC Processor Supplement (this document specifies the 32-bit ABI).		September, 1995
Advanced Encryption Standard (AES), Federal Information Processing Standards Publications, FIPS PUB 197.		2001
Logical Partitions on IBM PowerPC: A Guide to Working with LPAR on Power5 i5 Servers (This Redpaper can be downloaded from: http://www.redbooks.ibm.com/redbooks.nsf/RedpieceAbstracts/sg248000.html?Open)		February, 2005
Partitioning Implementations for IBM eServer™ p5 Servers (This can be downloaded from: http://www.redbooks.ibm.com/abstracts/sg247039.html)		February, 2005
Advanced Power Virtualization on IBM eserver p5 Servers (This Redpaper can be downloaded from: http://www.redbooks.ibm.com/abstracts/sg247940.html?Open)		December, 2005

Conventions and Notation

In this document, standard IBM notation is used, meaning that bits and bytes are numbered in ascending order from left to right. Thus, for a 4-byte word, bit 0 is the most-significant (high) bit and bit 31 is the least-significant (low) bit.



Notation for bit encoding is as follows:

- Hexadecimal values are preceded by x and enclosed in single quotation marks. For example: x‘0A00’.
- Binary values in sentences appear in single quotation marks. For example: ‘1010’.

The following software documentation conventions are used in this manual:

- Command (or instruction) names are written in **bold** type. For example: **put**, **get, rdch, wrch, rchcnt...**
- Variables are written in italic type. Required parameters are enclosed in angle brackets. Optional parameters are enclosed in brackets. For example: **get<*f,b*>[*s*]**.

- The notation <***f,b***> indicates that either the tag-specific fence or tag-specific barrier form is available for a referenced MFC command.

The following symbols are used in this document:

&	bitwise AND
|	bitwise OR
%	modulus
=	equal to
! =	not equal to
x ≥	greater than or equal to
x ≤	less than or equal to
x >> y	shift to the right; for example, 6 >> 2 = 1; least-significant y-bits are dropped
x << y	shift to the left; for example, 3 >> 2 = 12; least-significant y-bits are replaced by ‘0’s

Referencing Registers, Fields, and Bit Ranges

Registers are referenced by their full name or by their short name (also called the register mnemonic). Fields within registers are referenced by their full field name or by their field name. The field name or names are enclosed in brackets [ ].The following table describes how registers, fields, and bit ranges are referenced in this document and provides examples of the references.

Type of Reference	Format	Example
Reference to a specific register and a specific field using the register short name and the field name(s), bit number(s), or bit range.	Register_Short_Name[Bit_FieldName]	MSR[FE0]
	Register_Short_Name[Bit_Number]	MSR[52]
	Register_Short_Name[Field_Name1, Field_Name2]	MSR[FE0, FE1]
	Register_Short_Name[Bit_Number, Bit_Number]	MSR[52, 55]
	Register_Short_Name[Starting_Bit_Number:Ending_Bit_Number]	MSR[39:44]
Reference to a specific register and the setting for a specific field, bit, or range of bits using the register short name and the field name(s), bit number(s), or bit range that is followed by an equal sign (=) and a value that field.	Register_Short_Name[Field_Name] = ‘*n*’ (where *n* is a binary value for the bit or bit range)	MSR[FE0] = ‘1’
	Register_Short_Name[Field_Name] =x‘*n*’ (where *n* is a hexadecimal value for the bit or bit range)	MSR[FE] = x‘F’
	Register_Short_Name[Bit_Number] = ‘*n*’ (where *n* is a binary value for the bit or bit range)	MSR[52] = ‘0’
	Register_Short_Name[Bit_Number] = x‘*n*’ (where *n* is a hexadecimal value for the bit or bit range)	MSR[52] = xF’
	Register_Short_Name[Starting_Bit_Number:Ending_Bit_Number] = ‘*n*’ (where *n* is the binary or hex value for the bit or bit range)	MSR[39:43] = ‘10010’ [39:43] = ‘10010’
	Register_Short_Name[Starting_Bit_Number:Ending_Bit_Number] = ‘*n*’ (where *n* is the hexadecimal value for the field or bits)	MSR[39:43] = x‘11’ [39:43] = x‘11’
Note: The register short name is also called the register mnemonic.		

Terminology

In this handbook, a “memory-mapped I/O (MMIO)” register is any internal or external register that is accessed through the main-storage space with load and store instructions, whether or not the register is associated with an I/O controller or device.

Reserved Regions of Memory and Registers

Register locations that are marked as *reserved* and bit fields within registers that are marked as *reserved not implemented* operate in the same way: writes have no effect and reads return all ‘1’s.

Reserved areas of the MMIO-register memory map that are not assigned to any functional unit should not be read from or written to: doing so will cause serious errors in software as follows. For reads or writes generated from the SPE to unassigned reserved spaces, at least one of the following MFC_FIR[46,53,56,58,61] bits will be set and in most cases will cause a Checkstop. For reads or writes generated from the PPE to unassigned reserved spaces, at least one of the CIU_FIR[7,8] will be set and a Checkstop will occur. For reads or writes generated from the IOC to unassigned reserved spaces, the IOC will respond back to the IOIF device that sourced the address request with a ERR (error) response. No IOC FIR bits are set.

1. Overview of the Cell Broadband Engine Processor

This handbook presents both an overview and considerable detail about the extensive programming facilities of the Cell Broadband Engine (CBE) processor. The CBE processor is the first implementation of a new family of multiprocessors conforming to the *Cell Broadband Engine Architecture* (CBEA). The CBEA is a new architecture that extends the 64-bit *PowerPC Architecture*™. The CBEA and the CBE processor are the result of a collaboration between Sony, Toshiba, and IBM known as STI, formally begun in early 2001.

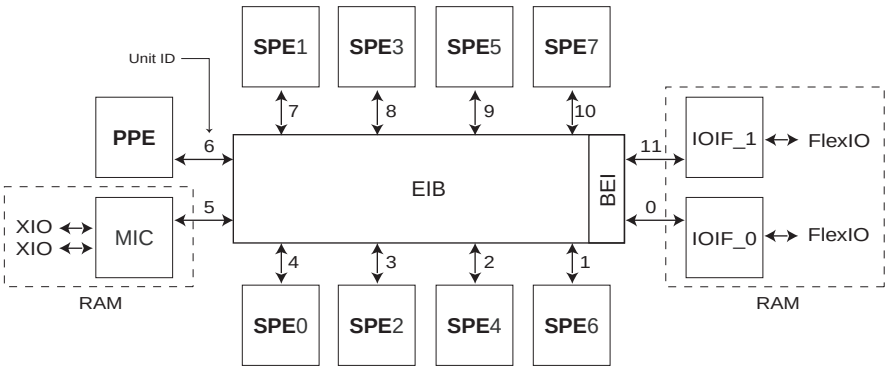
Although the CBE processor is initially intended for applications in media-rich consumer-electronics devices such as game consoles and high-definition televisions, the architecture has been designed to enable fundamental advances in processor performance. These advances are expected to support a broad range of applications in both commercial and scientific fields.

This handbook is written for the complete range of programmers, including those developing applications (user programs), libraries, device drivers, middleware, compilers, and operating systems. It assumes the reader is an experienced C/C++ programmer. It describes and presents examples of both basic and advanced programming concepts for single-instruction, multiple-data (SIMD) vector applications and the system software that supports such applications.

The handbook is system-independent, making no assumptions about development-tool or operating-system environments, other than the C/C++ language environment. The examples are chosen to highlight the general principals required for CBE-processor programming, such that an experienced programmer can apply this knowledge to their particular system environment.

Figure 1-1 shows a block diagram of the CBE-processor hardware. This figure is referred to later in this chapter and in subsequent chapters.

Figure 1-1. Cell Broadband Engine Overview



BEI	Cell Broadband Engine Interface	MIC	Memory Interface Controller
EIB	Element Interconnect Bus	PPE	PowerPC Processor Element
FlexIO	Rambus FlexIO Bus	RAM	Resource Allocation Management
IOIF	I/O Interface	SPE	Synergistic Processor Element
		XIO	Rambus XDR I/O (XIO) cell

1.1 Background

1.1.1 Motivation

The *Cell Broadband Engine Architecture* has been designed to support a very broad range of applications. The first implementation—the CBE processor, described in this handbook—is a single-chip multiprocessor with nine processor elements operating on a shared, coherent memory, as shown in *Figure 1-1* on page 33. In this respect, the CBE processor extends current trends in PC and server processors. The most distinguishing feature of the CBE processor is that, although all processor elements share memory, their function is specialized into two types: the Power Processor Element (PPE), and the Synergistic Processor Element (SPE). The CBE processor has one PPE and eight SPEs.

The first type of processor element, the PPE, contains a 64-bit PowerPC Architecture core. It complies with the 64-bit PowerPC Architecture and can run 32-bit and 64-bit operating systems and applications. The second type of processor element, the SPE, is optimized for running compute-intensive SIMD applications; it is not optimized for running an operating system. The SPEs are independent processor elements, each running their own individual application programs or threads. Each SPE has full access to coherent shared memory, including the memory-mapped I/O space. There is a mutual dependence between the PPE and the SPEs. The SPEs depend on the PPE to run the operating system, and, in many cases, the top-level thread control for an application. The PPE depends on the SPEs to provide the bulk of the application performance.

The SPEs are designed to be programmed in high-level languages, such as (but certainly not limited to) C/C++. They support a rich instruction set that includes extensive SIMD functionality. However, like conventional processors with SIMD extensions, use of SIMD data types is preferred, not mandatory. For programming convenience, the PPE also supports the standard PowerPC Architecture instructions and the Vector/SIMD Multimedia Extensions.

To an application programmer, the CBE processor looks like a 9-way coherent multiprocessor. The PPE is more adept than the SPEs at control-intensive tasks and quicker at task switching. The SPEs are more adept at compute-intensive tasks and slower than the PPE at task switching. However, either processor element is capable of both types of functions. This specialization is a significant factor accounting for the order-of-magnitude improvement in peak computational performance and chip-area-and-power efficiency that the CBE processor achieves over conventional PC processors.

The more significant difference between the SPE and PPE lies in how they access memory. The PPE accesses main storage (the effective-address space) with load and store instructions that move data between main storage and a private register file, the contents of which may be cached. The SPEs, in contrast, access main storage with Direct Memory Access (DMA) commands that move data and instructions between main storage and a private local memory, called a *local store* or *local storage* (LS). An SPE's instruction-fetches and load and store instructions access its private LS rather than shared main storage, and the LS has no associated cache. This three-level organization of storage (register file, LS, main storage), with asynchronous DMA transfers between LS and main storage, is a radical break from conventional architecture and programming models, because it explicitly parallelizes computation with the transfers of data and instructions that feed computation and store the results of computation in main storage.

One of the enabling accomplishments for this radical change is that memory latency, measured in processor cycles, has gone up several hundredfold from about the years 1980 to 2000. The result is that application performance is, in most cases, limited by memory latency rather than peak compute capability or peak bandwidth. When a sequential program on a conventional architecture performs a load instruction that misses in the caches, program execution comes to a halt for several hundred cycles. Compared to this penalty, the few cycles it takes to set up a DMA transfer for an SPE are a much better trade-off, especially considering the fact that each of the eight SPE's DMA controller can have up to 16 DMA transfer in flight simultaneously. Conventional processors, even with deep and costly speculation, manage to get, at best, a handful of independent memory accesses in flight.

One of the SPE's DMA transfer methods supports a list (such as a scatter-gather list) of DMA transfers that is constructed in an SPE's local store, so that the SPE's DMA controller can process the list asynchronously while the SPE operates on previously transferred data. In several cases, this approach to accessing memory has led to application performance exceeding that of conventional processors by almost two orders of magnitude—significantly more than one would expect from the peak performance ratio (about 10x) between the CBE processor and conventional PC processors. The DMA transfers can be set up and controlled by the SPE that is sourcing or receiving the data, or by the PPE or another SPE.

1.1.2 Power, Memory, and Frequency

The CBE processor overcomes three important limitations of contemporary microprocessor performance—power use, memory use, and clock frequency.

Microprocessor performance is approaching limits of power dissipation rather than integrated-circuit resources (transistors and wires). The only way to significantly increase the performance of microprocessors in this environment is to improve power-efficiency at about the same rate as the performance increase. The CBE processor does this by differentiating between the PPE, optimized to run an operating system and control-intensive code, and the eight SPEs, optimized to run compute-intensive applications. The control-plane PPE leaves the eight data-plane SPEs free to compute data-rich applications.

On today's symmetric multiprocessors—even those with integrated memory controllers—latency to DRAM memory is approaching 1,000 cycles. As a result, program performance is dominated by moving data between main storage and the processor. Compilers and application writers must manage this data movement explicitly, even though the hardware cache mechanisms are supposed to relieve them of this task. In contrast, the CBE processor's mechanisms for dealing with memory latencies—the three-level SPE memory structure (main storage, local stores, and large register files), and asynchronous DMA transfers—allow programmers to schedule simultaneous data and code transfers to cover long memory latencies. At 16 simultaneous transfers per SPE, the CBE processor can support up to 128 simultaneous transfers between the SPE local stores and main storage. This surpasses the bandwidth of conventional processors by a factor of almost twenty.

Conventional processors require increasingly deeper instruction pipelines to achieve higher operating frequencies. This technique has reached a point of diminishing returns—and even negative returns if power is taken into account. By specializing the PPE and the SPEs for control and compute-intensive tasks, respectively, these processing elements run at high frequencies without excessive overhead. The PPE achieves efficiency by executing two threads simultaneously rather than by optimizing single-thread performance. Each SPE achieves efficiency by

using a large register file that supports many simultaneous in-flight instructions without the overhead of register-renaming or out-of-order processing, and asynchronous DMA transfers, that support many concurrent memory operations without the overhead of speculation.

By distinguishing and separately optimizing control-plane and data-plane processor elements, the CBE processor mitigates the problems posed by power, memory, and frequency limitations. The net result is a multiprocessor that, at the power budget of a conventional PC processor, can provide approximately ten-fold the peak performance of a conventional processor. Of course, actual application performance varies. Some applications may benefit little from the SPEs, whereas others show a performance increase well in excess of ten-fold. In general, compute-intensive applications that use 32-bit or smaller data formats (such as single-precision floating-point and integer) are excellent candidates for the CBE processor.

1.1.3 Scope of this Handbook

As mentioned at the beginning, this handbook is written for the complete range of programmers, including those developing applications, libraries, device drivers, middleware, compilers, and operating systems.

The remainder of this chapter summarizes highlights of the hardware and programming environments. The two chapters that follow expand on these topics for the CBE processor's programming targets, the PPE and the SPEs.

This is followed in Chapters 4 through 16 with additional hardware overviews and details primarily relating to privilege-state (supervisor) programming. A few user-programming topics are also discussed in these chapters. The topics covered include virtual-memory management, cache management, I/O, resource-allocation management (the management of memory and I/O resources), interrupts, multithreading, hypervisors (the highest privilege state), SPE context-switching, the CBE clocks (time base) and decrementers, object-file formats, and power and performance monitoring.

Problem-state (user) programming topics are covered primarily in Chapters 17 through 24, and the appendices. The topics include the SPE channel interface, SPE events, DMA transfers and inter-processor communications, shared-storage synchronization, parallel programming techniques, SIMD programming techniques, PPE versus SPE support for vector operations, general SPE programming tips, and summaries of the PPE and SPE instruction sets and C/C++ intrinsics. A few supervisor-programming and compiler-writer topics are also covered in these chapters.

There are many programming examples throughout the handbook, and a rich set of active hyperlinks provide cross-references to relevant topics in various chapters. The *Glossary* on page 823 defines several hundred terms that have special meanings in the context of the CBE processor.

1.2 Hardware Environment

1.2.1 The Processor Elements

Figure 1-1 on page 33 shows a high-level block diagram of the CBE processor hardware. There is one PPE and eight identical SPEs. All processor elements are connected to each other and to the on-chip memory and I/O controllers by the memory-coherent Element Interconnect Bus (EIB).

The PPE contains a 64-bit, dual-thread *PowerPC Architecture* RISC core and supports a PowerPC virtual-memory subsystem. It has 32-KB Level-1 (L1) instruction and data caches and a 512-KB Level-2 (L2) unified (instruction and data) cache. It is intended primarily for control processing, running operating systems, managing system resources, and managing SPE threads. It can run legacy PowerPC-Architecture software and is well-suited to executing system-control code. The instruction set for the PPE is an extended version of the PowerPC instruction set. It includes the Vector/SIMD Multimedia Extensions and associated C/C++ intrinsic extensions.

The eight identical SPEs are Single-Instruction, Multiple-Data (SIMD) processor elements that are optimized for data-rich operations allocated to them by the PPE. Each SPE contains a RISC core, 256-KB software-controlled local store (LS) for instructions and data, and a 128-bit, 128-entry unified register file. The SPEs support a special SIMD instruction set—the *Synergistic Processor Unit Instruction Set Architecture*—and a unique set of commands for managing DMA transfers and interprocessor messaging and control. SPE DMA transfers access main storage using PowerPC effective addresses. As in the PPE, SPE address translation is governed by *PowerPC Architecture* segment and page tables, which are loaded into the SPEs by privileged software running on the PPE. The SPEs are not intended to run an operating system.

An SPE controls DMA transfers and communicates with the system by means of *channels* that are implemented in and managed by the SPE's Memory Flow Controller (MFC). The channels are unidirectional message-passing interfaces. The PPE and other devices in the system, including other SPEs, can also access this MFC state through the MFC's Memory-Mapped I/O (MMIO) registers and queues, which are visible to software in the main-storage address space.

For more information about the PPE, see *Section 2* on page 45. For more information about SPEs, see *Section 3* on page 63

1.2.2 Element Interconnect Bus

The Element Interconnect Bus (EIB) is the communication path for commands and data between all processor elements on the CBE processor and the on-chip controllers for memory and I/O. The EIB supports full memory-coherent and symmetric multiprocessor (SMP) operations. Thus, a CBE processor is designed to be ganged coherently with other CBE processors to produce a cluster.

The EIB consists of four 16-byte-wide data rings. Each ring transfers 128 bytes (one PPE cache line) at a time. Each processor element has one on-ramp and one off-ramp. Processor elements can drive and receive data simultaneously. *Figure 1-1* on page 33 shows the Unit ID numbers of each element and the order in which the elements are connected to the EIB. The connection order is important to programmers seeking to minimize the latency of transfers on the EIB:

latency is a function of the number of connection hops, such that transfers between adjacent elements have the shortest latencies and transfers between elements separated by six hops have the longest latencies.

The EIB's internal maximum bandwidth is 96 bytes per processor-clock cycle. Multiple transfers can be in-process concurrently on each ring, including more than 100 outstanding DMA memory requests between main storage and the SPEs. The EIB does not support any particular quality-of-service (QoS) behavior other than to guarantee forward progress. However, a Resource Allocation Management (RAM) facility, shown in *Figure 1-1* on page 33, resides in the EIB, and privileged software can use it to regulate the rate at which resource requestors (the PPE, SPEs, and I/O devices) can use memory and I/O resources.

1.2.3 Memory Interface Controller

The on-chip Memory Interface Controller (MIC) provides the interface between the EIB and physical memory. It supports one or two Rambus Extreme Data Rate (XDR) memory interfaces, which together support between 64 MB and 64 GB of XDR DRAM memory.

Memory accesses on each interface are 1 to 8, 16, 32, 64, or 128 bytes, with coherent memory-ordering. Up to 64 reads and 64 writes can be queued. The resource-allocation token manager provides feedback about queue levels.

The MIC has multiple software-controlled modes, including fast-path mode (for improved latency when command queues are empty), high-priority read (for prioritizing SPE reads in front of all other reads), early read (for starting a read before a previous write completes), speculative read, and slow mode (for power management). The MIC implements a closed-page controller (bank rows are closed after being read, written, or refreshed), memory initialization, and memory scrubbing.

The XDR DRAM memory is ECC-protected, with multi-bit error detection and optional single-bit error correction. It also supports write-masking, initial and periodic timing calibration, dynamic width control, sub-page activation, dynamic clock gating, and 4, 8, or 16 banks.

1.2.4 Cell Broadband Engine Interface Unit

The on-chip Cell Broadband Engine Interface (BEI) Unit supports I/O interfacing. It includes a Broadband Interface Controller (BIC), I/O Controller (IOC), and Internal Interrupt Controller (IIC), as defined in the *Cell Broadband Engine Architecture* document. It manages data transfers between the EIB and I/O devices and provides I/O address translation and command processing.

The BEI supports two Rambus FlexIO interfaces. One of the two interfaces (IOIF1) supports only a non-coherent I/O Interface (IOIF) protocol, which is suitable for I/O devices. The other interface (IOIF0, also called BIF/IOIF0) is software-selectable between the non-coherent IIOIF protocol and the memory-coherent Cell Broadband Engine Interface (BIF) protocol. The BIF protocol is the EIB's internal protocol. It can be used to coherently extend the EIB, through IOIF0, to another memory-coherent device, that can be another CBE processor.

1.3 Programming Environment

CBE-processor software development in the C/C++ language is supported by a rich set of language extensions that define C/C++ data types for SIMD operations and map C/C++ intrinsics (which are commands, in the form of function calls) to one or more assembly instructions. These language extensions, summarized in *Appendix A* on page 711 and *Appendix B* on page 761, give C/C++ programmers great control over code performance, without the need for assembly-language programming.

1.3.1 Instruction Sets

The instruction set for the PPE is an extended version of the *PowerPC Architecture* instruction set. The extensions consist of the Vector/SIMD Multimedia Extensions, a few additions and changes to *PowerPC Architecture* instructions, and C/C++ intrinsics for the Vector/SIMD Multimedia Extensions.

The instruction set for the SPEs is a new SIMD instruction set, the *Synergistic Processor Unit Instruction Set Architecture*, with accompanying C/C++ intrinsics, and a unique set of commands for managing DMA transfer, external events, interprocessor messaging, and other functions. The instruction set for the SPEs is similar to that of the PPE's Vector/SIMD Multimedia Extensions, in the sense that they operate on SIMD vectors. However, the two vector instruction sets are distinct, and programs for the PPE and SPEs are often compiled by different compilers.

Although most coding for the CBE processor will likely be done in a high-level language like C or C++, an understanding of the PPE and SPE machine instructions adds considerably to a software developer's ability to produce efficient, optimized code. This is particularly true because most of the C/C++ intrinsics have a mnemonic that relate directly to the underlying assembly-language mnemonic.

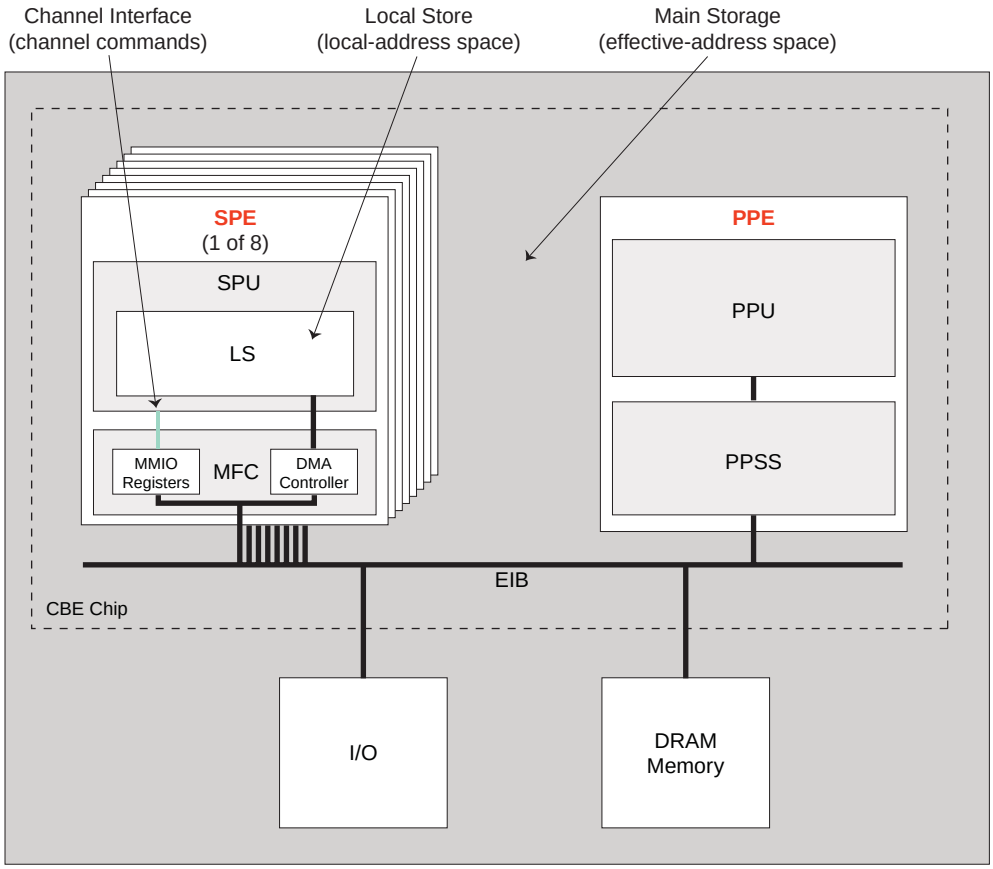
1.3.2 Storage Domains and Interfaces

The CBE processor has two types of storage domains—one *main-storage domain* and eight SPE *local-store (LS) domains*, as shown in *Figure 1-2* on page 40. In addition, each SPE has a *channel interface* for communication between its Synergistic Processor Unit (SPU) and its MFC. The main-storage domain, which is the entire effective-address space, can be configured by PPE privileged software to be shared by all processor elements and memory-mapped devices in the system[1]. An MFC's state is accessed by its associated SPU through the channel interface, and this state can also be accessed by the PPE and other devices (including other SPEs) by means of the MFC's MMIO registers in the main-storage space. An SPU's LS can also be accessed by the PPE and other devices through the main-storage space. The PPE accesses the main-storage space through its PowerPC Processor Storage Subsystem (PPSS).

The address-translation mechanisms used in the main-storage domain are described in *Section 4 Virtual Storage Environment* on page 79. The channel domain is described in *Section 19 DMA Transfers and Inter-Processor Communication* on page 505.

1. In the *PowerPC Architecture*, all I/O is memory-mapped.

Figure 1-2. Storage and Domains and Interfaces



DMA	Direct Memory Access	PPE	PowerPC Processor Element
EIB	Element Interconnect Bus	PPSS	PowerPC Processor Storage Subsystem
LS	Local Store	PPU	PowerPC Processor Unit
MFC	Memory Flow Controller	SPE	Synergistic Processor Element
MMIO	Memory-Mapped I/O	SPU	Synergistic Processor Unit

An SPE's SPU can fetch instructions only from its own LS, and load and store instructions executed by the SPU can only access the LS. SPU software uses LS addresses (not main-storage effective addresses) to do this. Each SPE's MFC contains a DMA controller. DMA-transfer requests contain both an LS address and an effective address, thereby facilitating transfers between the domains.

Data transfers between an SPE's LS and main storage are performed by the associated SPE, or by the PPE or another SPE, using the DMA controller in the MFC associated with the LS. Software running on the associated SPE interacts with its own MFC through its channel interface. The channels support enqueueing of DMA commands and other facilities, such as mailbox and signal-notification messages. Software running on the PPE or another SPE can interact with an MFC through MMIO registers, which are associated with the channels and visible in the main-storage space.

Each MFC maintains and processes two independent command queues for DMA and other commands—one queue for its associated SPU, and another queue for other devices accessing the SPE through the main-storage space. Each MFC can process multiple in-progress DMA commands. Each MFC can also autonomously manage a sequence of DMA transfers in response to a DMA list command from its associated SPU (but not from the PPE or other SPEs). Each DMA command is tagged with a Tag Group ID that allows software to check or wait on the completion of commands in a particular tag group.

The MFCs support naturally aligned DMA transfer sizes of 1, 2, 4, or 8 bytes, and multiples of 16 bytes, with a maximum transfer size of 16 KB per DMA transfer. DMA list commands can initiate up to 2,048 such DMA transfers. Peak transfer performance is achieved if both the effective addresses and the LS addresses are 128-byte aligned and the size of the transfer is an even multiple of 128 bytes.

Each MFC has a Synergistic Memory Management (SMM) unit that processes address-translation and access-permission information supplied by the PPE operating system. To process an effective address provided by a DMA command, the SMM uses essentially the same address-translation and protection mechanism used by the Memory Management Unit (MMU) in the PPE's PowerPC Processor Storage Subsystem (PPSS)[2]. Thus, DMA transfers are coherent with respect to system storage, and the attributes of system storage are governed by the page and segment tables of the *PowerPC Architecture*.

1.3.3 Byte Ordering and Bit Numbering

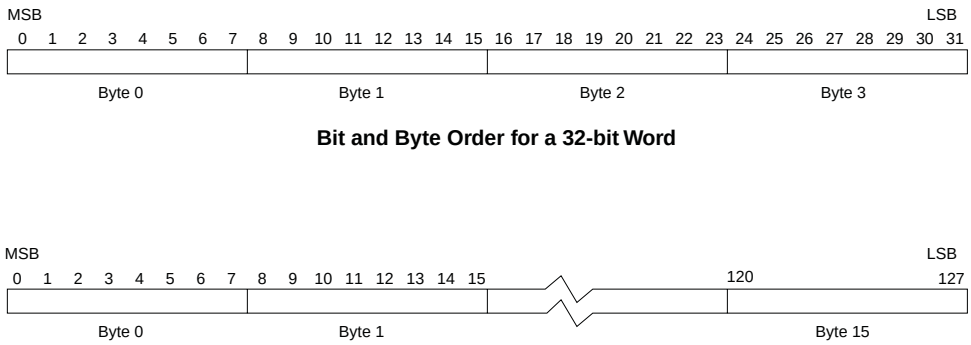
Storage of data and instructions in the CBE processor uses big-endian ordering, which has the following characteristics:

- Most-significant byte stored at the lowest address, and least-significant byte stored at the highest address.
- Bit numbering within a byte goes from most-significant bit (bit 0) to least-significant bit (bit n). This differs from some other big-endian processors.

A summary of the byte- and bit-ordering in memory, as well as the bit-numbering conventions, is shown in *Figure 1-3*.

2. See *Section 4.3.6* on page 116 for the different way in which the SMM translates real addresses, as compared with the MMU.

Figure 1-3. Big-Endian Byte and Bit Ordering



Neither the PPE nor the SPEs, including their MFCs, support little-endian byte-ordering. The MFC's DMA transfers are simply byte moves, without regard to the numerical significance of any byte. Thus, the big-endian or little-endian issue becomes irrelevant to the actual movement of a block of data. The byte-order mapping only becomes significant when data is loaded or interpreted by a processor element or an MFC.

1.3.4 Runtime Environment

The PPE runs *PowerPC Architecture* applications and operating systems, which may include both *PowerPC Architecture* instructions and Vector/SIMD Multimedia Extension instructions. To use all of the CBE processor's features, the PPE requires an operating system that supports these features, such as multiprocessing with the SPEs, access to the PPE Vector/SIMD Multimedia Extension operations, the CBE interrupt controller, and all the other functions provided by the CBE processor. This handbook is system-independent, in the sense of making no assumptions about specific operating-system or development-tool environments.

It is common to run a main program on the PPE that allocates threads to the SPEs. In such an application, the *main thread* is said to spawn one or more *CBE tasks*. A CBE task has one or more main threads associated with it, along with some number of *SPE threads*. An SPE thread is a thread that is spawned to run on an available SPE. These terms are defined in *Table 1-1*. The software threads are unrelated to the hardware multithreading capability of the PPE.

Table 1-1. Definition of Threads and Tasks

Term	Definition
Main Thread	A thread running on the PPE.
CBE Task	A task running on the PPE and SPE. Each such task: • Has one or more main threads and some number of SPE threads. • All the main threads within the task share the task's resources, including access to the SPE threads.
SPE Thread	A thread running on an SPE. Each such thread: • Has its own 128 x 128-bit register file, program counter and MFC-DMA command queues. • Can communicate with other execution units (or with main storage through the MFC channel interface).

A main thread can interact directly with an SPE thread through the SPE's LS. It can interact indirectly through the main-storage space. A thread can poll or sleep, waiting for SPE threads.

The operating system defines the mechanism and policy for selecting an available SPE. It must prioritize among all the CBE applications in the system, and it must schedule SPE execution independently from regular main threads. The operating system is also responsible for run-time loading, passing parameters to SPE programs, notification of SPE events and errors, and debugger support.

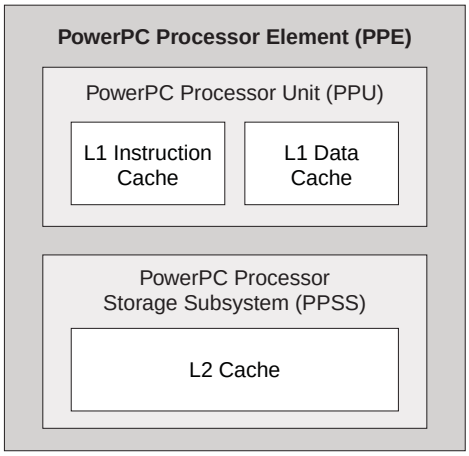
The next two chapters provide an expanded overview of the hardware and programming environments for the PPE (*Section 2* on page 45) and the SPEs (*Section 3* on page 63).

2. PowerPC Processor Element

The PowerPC Processor Element (PPE) is a general-purpose, dual-threaded, 64-bit RISC processor that conforms to the *PowerPC Architecture*, version 2.02, with the Vector/SIMD Multi-media Extensions. Programs written for the PowerPC 970 processor, for example, should run on the CBE processor without modification.

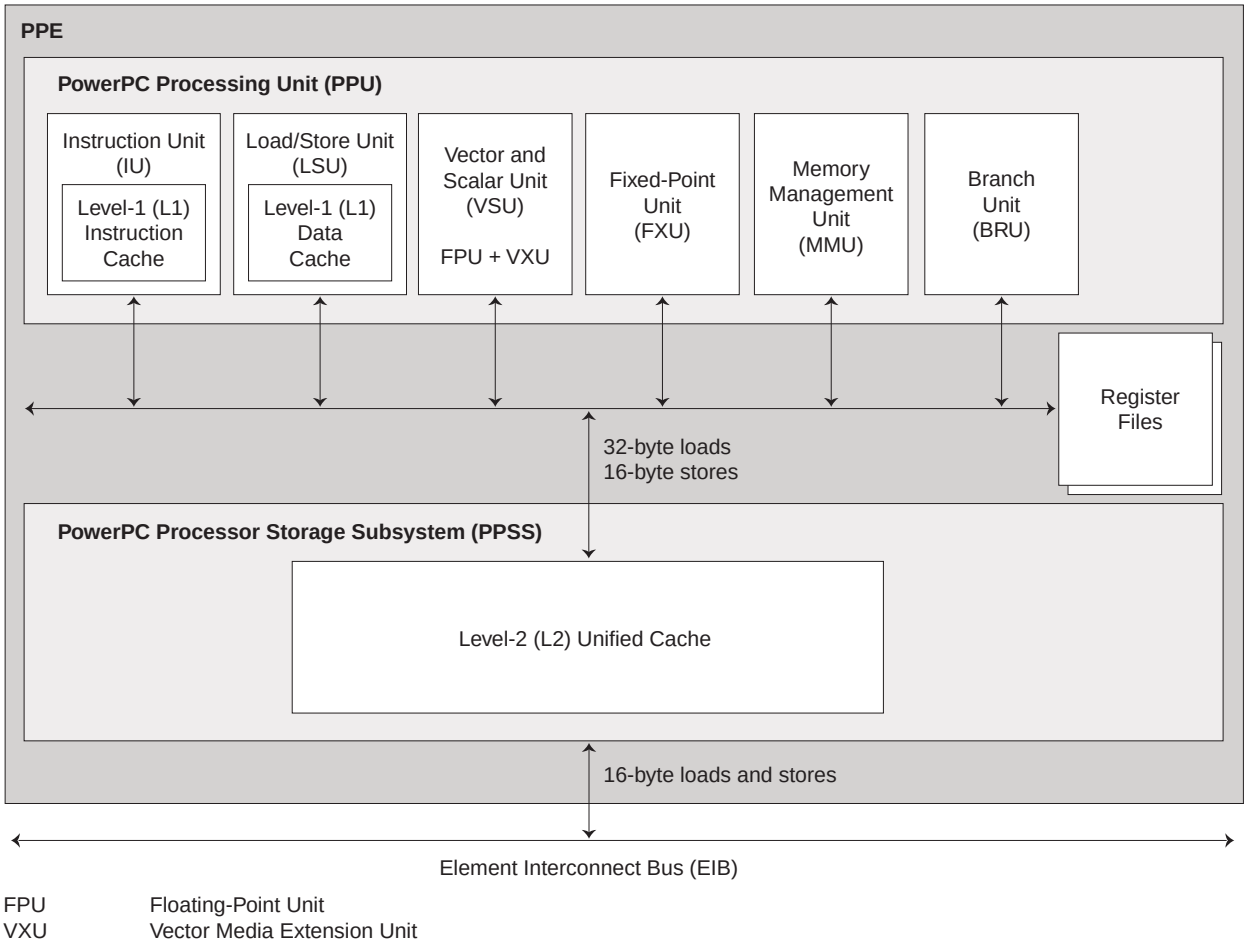
The PPE is responsible for overall control of a CBE system. It runs the operating system(s) for all applications running on the PPE and SPEs. The PPE consists of two main units, the PowerPC Processor Unit (PPU) and the PowerPC Processor Storage Subsystem (PPSS), shown in *Figure 2-1*.

Figure 2-1. PPE Block Diagram



The PPU performs instruction execution. Its has a level-1 (L1) instruction cache and data cache and six execution units. It can load 32 bytes and store 16 bytes, independently and memory-coherently, per processor cycle. The PPSS handles memory requests from the PPU and external requests to the PPE from SPEs or I/O devices. It has a unified level-2 (L2) instruction and data cache. The PPU and the PPSS and their functional units are shown in *Figure 2-2* on page 46. The sections that follow describe their functions.

Figure 2-2. PPE Functional Units



FPU Floating-Point Unit
VXU Vector Media Extension Unit

2.1 PowerPC Processor Unit

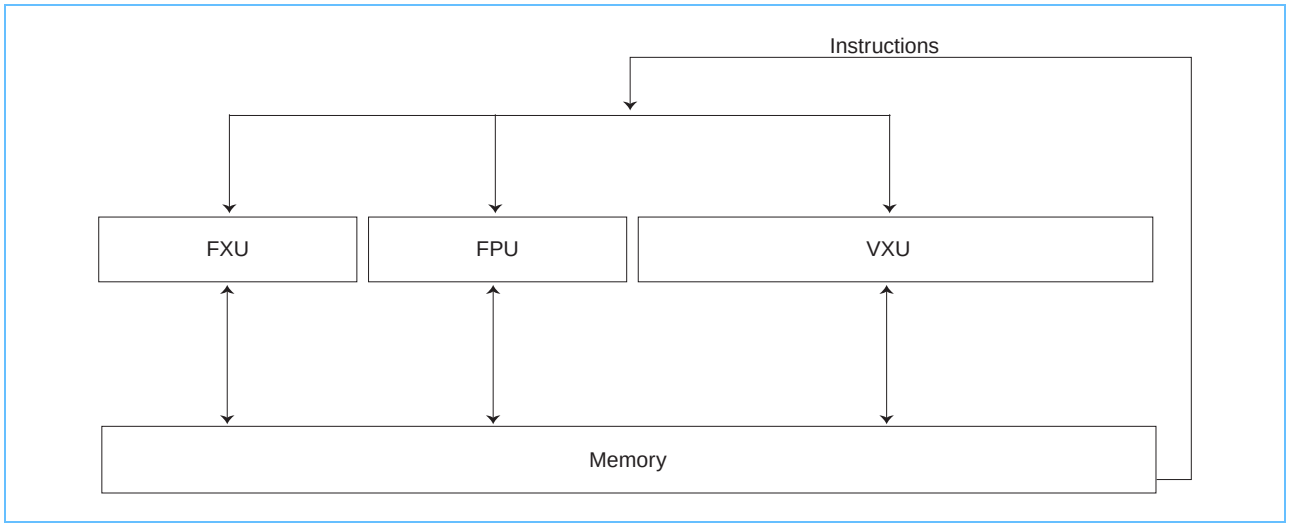
The PPU executes the *PowerPC Architecture* instruction set and the Vector/SIMD Multimedia Extension instructions. It has duplicate sets of the PowerPC and vector user-state register files (one set for each thread) plus one set of the following functional units:

- *Instruction Unit (IU)*—The IU performs the instruction-fetch, decode, dispatch, issue, branch, and completion portions of execution. It contains the L1 instruction cache, which is 32 KB, 2-way set-associative, reload-on-error, and parity protected. The cache-line size is 128 bytes.
- *Load and Store Unit (LSU)*—The LSU performs all data accesses, including execution of load and store instructions. It contains the L1 data cache, which is 32 KB, 4-way set-associative, write-through, and parity protected. The cache-line size is 128 bytes.
- *Vector/Scalar Unit (VSU)*—The VSU includes a Floating-Point Unit (FPU) and a 128-bit Vector/SIMD Multimedia Extension Unit (VXU), which together execute floating-point and Vector/SIMD Multimedia Extension instructions.

- *Fixed-Point Unit (FXU)*—The FXU executes fixed-point (integer) operations, including add, multiply, divide, compare, shift, rotate, and logical instructions.
- *Memory Management Unit (MMU)*—The MMU manages address translation for all memory accesses. It has a 64-entry Segment Lookaside Buffer (SLB) and 1024-entry, unified, parity-protected Translation Lookaside Buffer (TLB). It supports three simultaneous page sizes—4 KB, plus two sizes selectable from 64 KB, 1 MB, or 16 MB.

The 128-bit VXU operates concurrently with the FXU and the FPU, as shown *Figure 2-3* and subject to limitations described in *Appendix A.5* on page 750. All Vector/SIMD Multimedia Extension instructions are designed to be easily pipelined. Parallel execution with the fixed-point and floating-point instructions is simplified by the fact that Vector/SIMD Multimedia Extension instructions do not generate exceptions (other than data-storage interrupts on loads and stores), do not support unaligned memory accesses or complex functions, and share few resources or communication paths with the other PPE execution units. A brief summary of the PPE instructions, the units on which they execute, and their latencies is given in *Table 2-2* on page 53 and *Table 2-3* on page 58.

Figure 2-3. Concurrent Execution of Fixed-Point, Floating-Point, and Vector Extension Units



The MMU manages virtual-memory address translation and memory protection in compliance with the 64-bit *PowerPC Architecture*. The MMU supports the following address-space sizes:

- *Real Address (RA) Space*—2^{42} bytes. These are the addresses, in real storage or on an I/O device, whose physical addresses have been mapped by privileged PPE software to the RA space. Real storage can include on-chip SPE LS memory and off-chip memory and I/O devices.
- *Effective Address (EA) Space*—2^{64} bytes. These are the addresses generated by programs. *Figure 1-2 Storage and Domains and Interfaces* on page 40 illustrates the effective address space.
- *Virtual Address (VA) Space*—2^{65} bytes. These are the addresses used by the MMU in the PPE and in the MFC of each SPE to translate between EAs and RAs.

2.2 PowerPC Processor Storage Subsystem

The PPSS handles all memory accesses by the PPU and memory-coherence (snooping) operation from the Element Interconnect Bus (EIB). The PPSS has a unified, 512-KB, 8-way set-associative, write-back L2 cache with error-correction code (ECC). Like the L1 caches, the cache-line size for the L2 is 128 bytes. The cache has a single-port read/write interface to main storage that supports eight software-managed data-prefetch streams. It includes the contents of the L1 data cache but is not guaranteed to contain the contents of the L1 instruction cache, and it provides fully coherent symmetric multiprocessor (SMP) support.

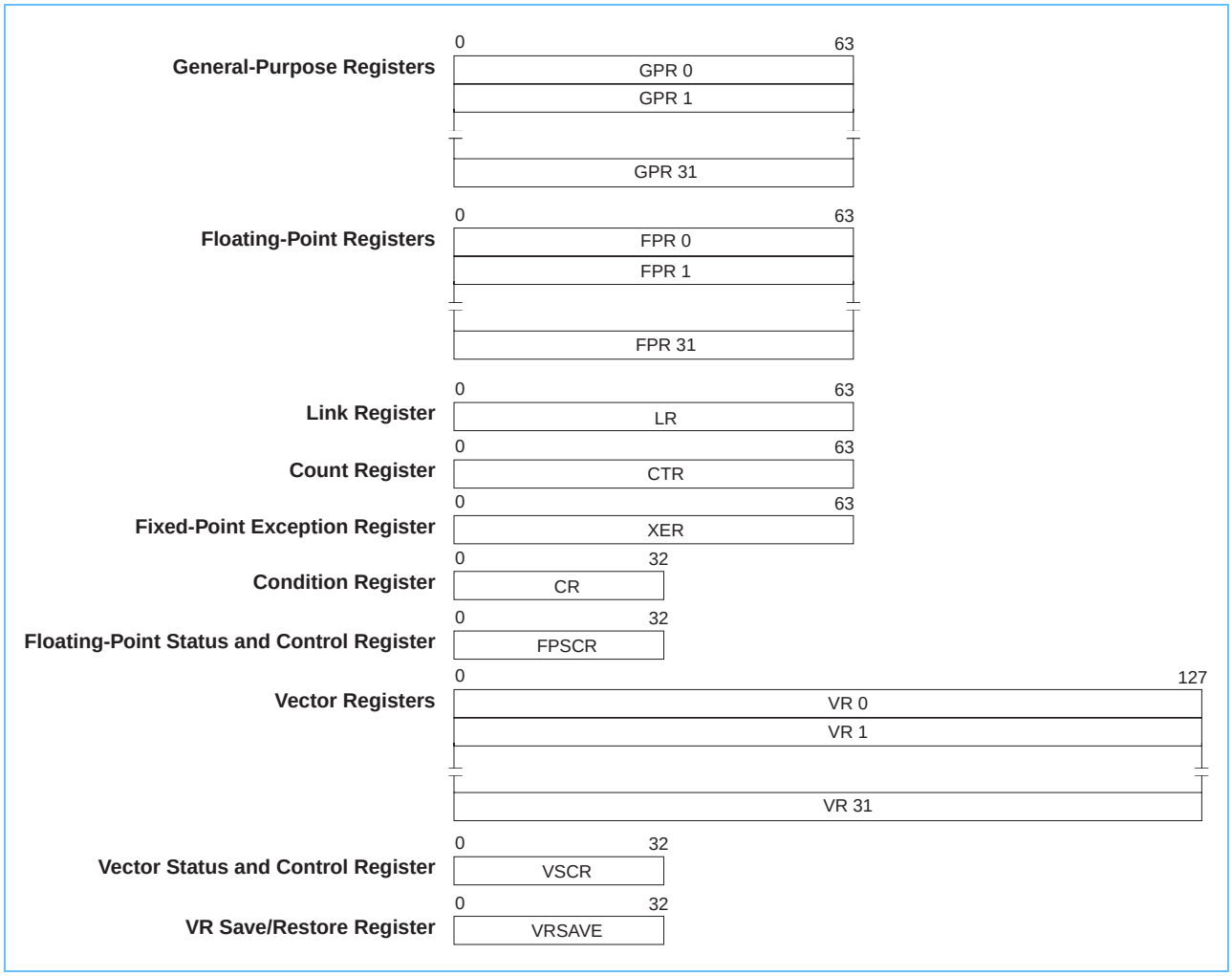
The PPSS performs data-prefetch for the PPU and bus arbitration and pacing onto the EIB. Traffic between the PPU and PPSS is supported by a 32-byte load port (shared by MMU, L1 instruction cache, and L1 data cache requests), and a 16-byte store port (shared by MMU and L1 data cache requests).

The interface between the PPSS and EIB supports 16-byte load and 16-byte store buses. One storage access occurs at a time, and all accesses appear to occur in program order. The interface supports Resource Allocation Management (*Section 8* on page 203), which allows privileged software to control the amount of time allocated to various resource groups. The L2 cache and the TLB use Replacement Management Tables (*Section 6.3* on page 154), which allow privileged software to control the use of the L2 and TLB. This software control over cache and TLB resources is especially useful for real-time programming.

2.3 PPE Registers

The PPE problem-state (user) registers are shown in *Figure 2-4* on page 49. All computational instructions operate on registers; no computational instructions modify main storage. To use a storage operand in a computation and then modify the same or another storage location, the contents of the storage operand must be loaded into a register, modified, and then stored back to the target location.

Figure 2-4. PPE User Register Set



The PPE registers include:

- *General-Purpose Registers (GPRs)*—The 32 GPRs are 64 bits wide. Fixed-point instructions operate on the full 64 bits. These instructions are mode-independent, except that in 32-bit mode[3], the processor uses only the low-order 32 bits for determining a memory address and the carry, overflow, and record status bits.
- *Floating-Point Registers (FPRs)*—The 32 FPRs are 64 bits wide. The internal format of floating-point data is the IEEE 754 double-precision format. Single-precision results are maintained internally in the double-precision format.
- *Link Register (LR)*—The 64-bit LR can be used to hold the effective address of a branch target. Branch instructions with the link bit (LK) set to 1 (that is, subroutine-call instructions) copy the next instruction address into the LR. The contents of a GPR can be copied to or from the LR using a Move To Special-Purpose Register instruction or Move From Special-Purpose Register instruction, respectively.

3. Machine State Register `MSR[SF]` bit = ‘0’.

- *Count Register (CTR)*—The 64-bit CTR can be used to hold either a loop counter or the effective address of a branch target. Some conditional-branch instruction forms decrement the CTR and test it for a ‘0’ value. The contents of a GPR can be copied to or from the CTR using a Move To Special-Purpose Register instruction or Move From Special-Purpose Register instruction, respectively.
- *Fixed-Point Exception Register (XER)*—The 64-bit XER contains the carry and overflow bits and the byte count for the move-assist instructions. Most arithmetic operations have instruction forms for setting the carry bit and overflow bit.
- *Condition Register (CR)*—Conditional comparisons are performed by first setting a condition code in the 32-bit CR with a compare instruction or with a recording instruction. The condition code is then available as a value or can be tested by a branch instruction to control program flow. The CR consists of eight independent 4-bit fields grouped together for convenient save or restore during a context switch. Each field can hold status information from a comparison, arithmetic, or logical operation. The compiler can schedule CR fields to avoid data hazards in the same way that it schedules use of GPRs. Writes to the CR occur only for instructions that explicitly request them; most operations have recording and nonrecording instruction forms.
- *Floating-Point Status and Control Register (FPSCR)*—The processor updates the 32-bit FPSCR after every PowerPC (but not Vector/SIMD Multimedia Extension) floating-point operation to record information about the result and any associated exceptions. The status information required by IEEE 754 is included, plus some additional information for exception handling.
- *Vector Registers (VRs)*—There are 32 128-bit-wide VRs. They serve as source and destination registers for all vector instructions.
- *Vector Status and Control Register (VSCR)*—The 32-bit VSCR is read and written in a manner similar to the FPSCR. It has 2 defined bits, a non-Java™ mode bit and a saturation bit; the remaining bits are reserved. Special instructions are provided to move the VSCR from and to a VR register.
- *Vector Save Register (VRSAVE)*—The 32-bit VRSAVE register assists user and privileged software in saving and restoring the architectural state across context switches.

The PPE hardware supports two simultaneous threads of execution. All architected and special-purpose registers are duplicated, except those that deal with system-level resources such as logical partitions (*Section 11* on page 327), memory, and thread-control (*Section 10* on page 295). Most nonarchitected resources, such as caches and queues, are shared by both threads, except in cases where the resource is small or offers a critical performance improvement to multithreaded applications. Because of this duplication of state, the PPE can be viewed as a 2-way multiprocessor with shared dataflow. The two hardware-thread environments appear to software as two independent processors.

For details on the complete set of PPE registers—both problem-state (user) and privilege-state (supervisor)—see the following documents:

- *Cell Broadband Engine Registers*.
- *PowerPC Microprocessor Family: The Programming Environments for 64- and 32-Bit Microprocessors*.
- *PowerPC Microprocessor Family: Vector/SIMD Multimedia Extension Technolgy Programming Environments Manual*.

2.4 PowerPC Instructions

The PPE's instruction set is compatible with version 2.0.2 of the *PowerPC Architecture* instruction set, with Vector/SIMD Multimedia Extensions, a few new instructions, new meaning for a few existing *PowerPC Architecture* instructions, and implementation of various optional *PowerPC Architecture* instructions. C/C++ intrinsics for the Vector/SIMD Multimedia Extension instructions are also provided.

This section gives a brief overview of the *PowerPC Architecture* instructions. *Section 2.5* on page 55 gives an overview of the Vector/SIMD Multimedia Extension instructions, and *Section 2.6* on page 59 gives an overview of the C/C++ intrinsics. *Appendix A PPE Instruction Set and Intrinsics* on page 711 gives a more complete summary of the instructions and intrinsics, including details about new instructions, compatibility with *PowerPC Architecture* instructions, issue rules, pipeline stages, and compiler optimizations.

2.4.1 Data Types

The PowerPC instruction set defines the data types shown in *Table 2-1*. Additional 128-bit vector and scalar data types defined in the Vector/SIMD Multimedia Extensions are described in *Section 2.5.2* on page 57. For details about alignment and performance characteristics for misaligned loads, see *Table A-2 Storage Alignment for PowerPC Instructions* on page 722.

Table 2-1. PowerPC Data Types

Data Types	Width (Bits)
Fixed-Point	
Byte (signed and unsigned)	8
Halfword (signed and unsigned)	16
Word (signed and unsigned)	32
Doubleword (signed and unsigned)	64
Floating-Point	
Single-precision	32
Double-precision	64

2.4.2 Addressing Modes

Whenever instruction addresses are presented to the processor, the low-order 2 bits are ignored. Similarly, whenever the processor develops an instruction address, the low-order 2 bits are '0'. The address of either an instruction or a multiple-byte data value is its lowest-numbered byte. This address points to the most-significant byte (the big-endian convention).

All instructions, except branches, generate addresses by incrementing a program counter. For load and store instructions that specify a base register, the effective address in memory for a data value is calculated relative to the base register in one of three ways:

- *Register* + *Displacement*—The displacement (D) forms of the load and store instructions form the sum of a displacement specified by the sign-extended 16-bit immediate field of the instruction plus the contents of the base register.

- *Register* + *Register*—The indexed (X) forms of the load and store instructions form the sum of the contents of the index register plus the contents of the base register.
- *Register*—The Load String Immediate and Store String Immediate instructions use the unmodified contents of the base register.

Loads and stores can specify an update form that reloads the base register with the computed address. Loads and stores also have byte-reverse forms.

Branches are the only instructions that explicitly specify the address of the next instruction. A branch instruction specifies the effective address of the branch target in one of the following ways:

- *Branch Not Taken*—The byte address of the next instruction is the byte address of the current instruction, plus 4.
- *Absolute*—Branch instructions to absolute addresses transfer control to the word address given in an immediate field of the branch instruction.
- *Relative*—Branch instructions to relative addresses transfer control to the word address given by the sum of the immediate field of the branch instruction and the word address of the branch instruction itself.
- *Link Register* or *Count Register*—The Branch Conditional to Link Register and Branch Conditional to Count Register instructions transfer control to the effective byte address of the branch target specified in the Link Register or Count Register, respectively.

2.4.3 Instructions

All PowerPC instructions are four bytes long and aligned on word (4-byte) boundaries. Most instructions can have up to three operands. Most computational instructions specify two source operands and one destination operand. Signed integers are represented in twos-complement form.

The instructions include the following types:

- *Load and Store Instructions*—These include fixed-point and floating-point load and store instructions, with byte-reverse, multiple, and string options for the fixed-point loads and stores. The fixed-point loads and stores support byte, halfword, word, and doubleword operand accesses between storage and the 32 general-purpose registers (GPRs). The floating-point loads and stores support word and doubleword operand accesses between storage and the 32 floating-point registers (FPRs). The byte-reverse forms have the effect of loading and storing data in little-endian order, although the CBE processor does not otherwise support little-endian order.
- *Fixed-Point Instructions*—These include arithmetic, compare, logical, and rotate/shift instructions. They operate on byte, halfword, word, and doubleword operands.
- *Floating-Point Instructions*—These include floating-point arithmetic, multiply-add, compare, and move instructions, as well as instructions that affect the floating-point status and control register (FPSCR). Floating-point instructions operate on single-precision and double-precision floating-point operands.
- *Memory Synchronization Instructions*—These 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. The instruction

types include load and store with reservation, synchronization, and enforce in-order execution of I/O. They are especially useful for multiprocessing. For details about synchronization, see *Section 20 Shared-Storage Synchronization* on page 551.

- *Flow Control Instructions*—These include branch, Condition-Register logical, trap, and other instructions that affect the instruction flow.
- *Processor Control Instructions*—These instructions are used for synchronizing memory accesses and managing caches, TLBs, segment registers, and other privileged processor state. They include move-to/from special-purpose register instructions.
- *Memory and Cache Control Instructions*—These instructions control caches, TLBs, and segment registers.

Table 2-2 summarizes the PowerPC instructions, latencies, throughput, and execution units. The Number of Instructions column includes instruction versions that are recording (those that not only perform the stated function but also set bits in the Condition Register to reflect characteristic of the result of execution). A much more detailed list of the instructions is given in *Table A-1* on page 712. For details of the instruction pipeline stages, see *Appendix A.6 Pipeline Stages* on page 752. For a complete description of the PowerPC instructions, see the *PowerPC Microprocessor Family: Programming Environments Manual for 64- and 32-Bit Microprocessors*.

Table 2-2. PowerPC Instruction Summary (Page 1 of 3)

Description	Number of Instructions	Latency (cycles)[1]	Throughput (IPC)[1,2]	Execution Unit[3]
Arithmetic	59	1 or 2	1 per latency cycle	FXU
Branch	12	1 or 2	1 per latency cycle	BRU
Compare	5	1	1 per latency cycle	FXU
Count Leading Zeros Recording	2	See *Table A-3* on page 724	1 per latency cycle	FXU
Count Leading Zeros Population Count	3	2	1 per latency cycle	FXU
Data-Cache Control	5	N/A	1 per latency cycle	LSU
Divide	8	10 to 70	1 per 10 to 70 latency cycles	FXU
Divide Recording	8	See *Table A-3* on page 724	1 per latency cycle	FXU
Extend Sign	3	2	1 per latency cycle	FXU
Extend Sign Recording	3	See *Table A-3* on page 724	1 per latency cycle	FXU
Floating-Point Compare	2	1	1 per latency cycle	FPU
Floating-Point Divide (IEEE)	2	74	1 per 74 latency cycles	FPU
Floating-Point Divide (non-IEEE)[4]	2	56	1 per 56 latency cycles	FPU
Floating-Point Divide Recording	2	11	1 per latency cycle	FPU
Floating-Point Move Floating-Point Arithmetic	20	10 or 11	1 per latency cycle	FPU
Floating-Point Multiply	16	10 or 11	1 per latency cycle	FPU
Floating-Point Reciprocal	3	10 or 11	1 per latency cycle	FPU

Table 2-2. PowerPC Instruction Summary (Page 2 of 3)

Description	Number of Instructions	Latency (cycles)[1]	Throughput (IPC)[1,2]	Execution Unit[3]
Floating-Point Round or Convert	12	10 or 11	1 per latency cycle	FPU
Floating-Point Select	2	10 or 11	1 per latency cycle	FPU
Floating-Point Square Root (IEEE)	2	84	1 per 84 latency cycles	FPU
Floating-Point Square Root (non-IEEE)[4]	2	66	1 per 66 latency cycles	FPU
Floating-Point Square Root Recording	3	11	1 per latency cycle	FPU
Instruction synchronize	1	N/A		LSU
Instruction-Cache Block invalidate	1	N/A		LSU
Load	18	2	1 per latency cycle	LSU
Load Algebraic	7	See *Table A-4* on page 726	1 per latency cycle	LSU, FXU
Load and Reserve	2	N/A		LSU
Load Floating-Point	8	1 or 2	1 per latency cycle	LSU, FXU, or FPU
Load Multiple Word Load String Word (aligned)	3	See *Table A-4* on page 726	1 per latency cycle	LSU
Logical	12	2	1 per latency cycle	FXU
Logical Recording	11	See *Table A-3* on page 724	1 per latency cycle	FXU
Conditional Register (CR) logical operations	8	1	1 per latency cycle	BRU
Move to/from non-GPR/FPR registers	24	See *Table A-1* on page 712		FXU, FPU, or BRU
Multiply Immediate	9	6 to 15	1 per 6 to 15 latency cycles	FXU
Multiply Recording	8	See *Table A-3* on page 724	1 per latency cycle	FXU
Rotate Immediate Shift Immediate	8	2	1 per latency cycle	FXU
Rotate Recording Rotate Clear	13	See *Table A-3* on page 724	1 per latency cycle	FXU
Shift	14	See *Table A-3* on page 724	1 per latency cycle	FXU
SLB Move	3	N/A		LSU
SLB or TLB Invalidate or Synchronize	5	N/A		LSU
Store	17	N/A	1 per latency cycle	LSU, FXU
Store Conditional	2	N/A		LSU, FXU
Store Floating-Point	5	N/A	1 per latency cycle	LSU, FPU
Store Floating-Point with Update	4	2	1 per latency cycle	LSU, FPU, FXU

Table 2-2. PowerPC Instruction Summary (Page 3 of 3)

Description	Number of Instructions	Latency (cycles)[1]	Throughput (IPC)[1,2]	Execution Unit[3]
Store Multiple Word Store String Word	3	N/A	1 per latency cycle	LSU, FXU
Synchronize	3	N/A		LSU
System call Return from interrupt	3	N/A		IU
Trap	4	N/A	1 per latency cycle	FXU

1. Throughput is after startup. Some fields are labelled “N/A” to indicate that a more elaborate description is required (and is best understood by taking a broader view of the machine as a whole), This includes complex microcoded instructions. Instructions that do not modify a register are also labelled “N/A.” See *Table A-1* on page 712 for details.
2. IPC is Instructions Per Cycle.
3. The “Execution Units” are BRU: Branch unit, IU: Instruction unit, FPU: Floating-point unit, FXU: Fixed-point unit, LSU: Load and store unit. See *Figure 2-2* on page 46.
4. Non-IEEE results are accurate to within 1 or 2 ULPs of the correctly rounded IEEE result, depending on rounding mode: 1 ULP for to-nearest and to-zero rounding modes (the most common), 2 ULP for the other modes.

2.5 Vector/SIMD Multimedia Extension Instructions

Vector/SIMD Multimedia Extension instructions can be freely mixed with PowerPC instructions in a single program. The 128-bit Vector/SIMD Multimedia Extension unit (VXU) operates concurrently with the PPU’s 32-bit fixed-point unit (FXU) and 64-bit floating-point unit (FPU). Like PowerPC instructions, the Vector/SIMD Multimedia Extension instructions are four bytes long and word-aligned. The Vector/SIMD Multimedia Extension instructions support simultaneous execution on multiple elements that make up the 128-bit vector operands.

Vector/SIMD Multimedia Extension instructions do not generate exceptions (other than data-storage interrupt exceptions on loads and stores), do not support unaligned memory accesses or complex functions, and share few resources or communication paths with the other PPE execution units.

The sections that follow briefly summarize features of the extension. For a more detailed list of instructions, see *Appendix A.3* on page 737. For a complete description of the PowerPC instruction set, see *PowerPC Microprocessor Family: Vector/SIMD Multimedia Extension Technolgy Programming Environments Manual*.

2.5.1 SIMD Vectorization

A *vector* is an instruction operand containing a set of data elements packed into a one-dimensional array. The elements can be fixed-point or floating-point values. Most Vector/SIMD Multimedia Extension and SPU instructions operate on vector operands. Vectors are also called Single-Instruction, Multiple-Data (*SIMD) operands*, or *packed operands*.

SIMD processing exploits data-level parallelism. Data-level parallelism means that the operations required to transform a set of vector elements can be performed on all elements of the vector at the same time. That is, a single instruction can be applied to multiple data elements in parallel.

Support for SIMD operations is pervasive in the CBE processor. In the PPE, they are supported by the Vector/SIMD Multimedia Extension instructions, described in the sections immediately below. In the SPEs, they are supported by the SPU instruction set, described in *Section 3.3* on page 74.

In both the PPE and SPEs, vector registers hold multiple data elements as a single vector. The data paths and registers supporting SIMD operations are 128 bits wide, corresponding to four full 32-bit words. This means that four 32-bit words can be loaded into a single register, and, for example, added to four other words in a different register in a single operation. *Figure 2-5* shows such an operation. Similar operations can be performed on vector operands containing 16 bytes, 8 halfwords, or 2 doublewords.

Figure 2-5. Four Concurrent Add Operations

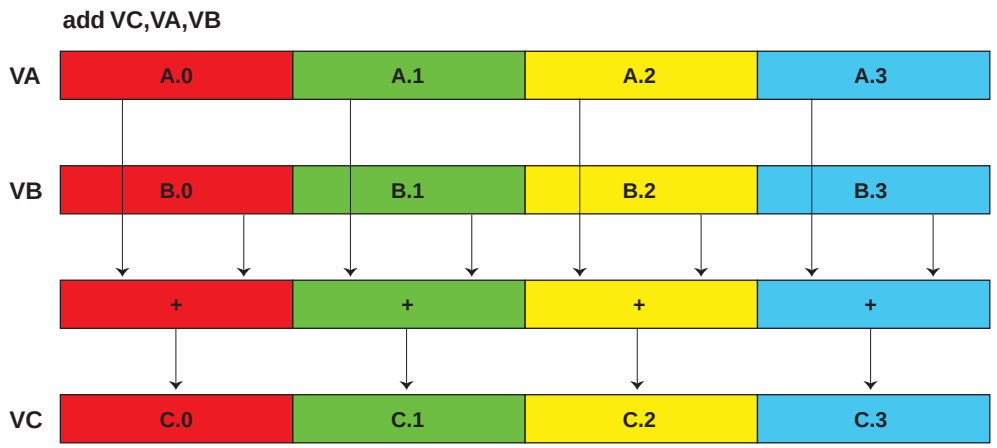
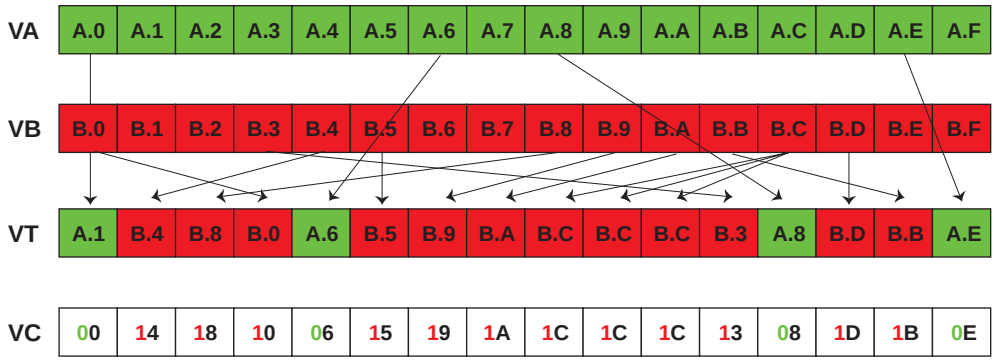


Figure 2-6 shows another example of a SIMD operation—in this case, a byte-shuffle (permute) operation. Here, the bytes selected for the shuffle from the source registers, VA and VB, are based on byte entries in the control vector, VC, in which a 0 specifies VA and a 1 specifies VB. The result of the shuffle is placed in register VT. The shuffle operation is extremely powerful and finds its way into many applications in which data reordering, selection, or merging is required.

Figure 2-6. Byte-Shuffle (Permute) Operation



The process of preparing a program for use on a vector processor is called *vectorization* or *SIMDization*. It can be done manually by the programmer, or it can be done by a compiler that does *auto-vectorization*. See *Section 22 SIMD Programming* on page 619 for details.

2.5.2 Data Types

The operands for most Vector/SIMD Multimedia Extension instructions are quadword (128-bit) vectors. These vectors can contain elements that are (a) fixed-point bytes, halfwords, or words, or (b) floating-point words.

The vector registers are 128 bits wide and can contain:

- Sixteen 8-bit values, signed or unsigned.
- Eight 16-bit values, signed or unsigned.
- Four 32-bit values, signed or unsigned.
- Four single-precision IEEE-754 floating point values.

The C/C++ intrinsics for the Vector/SIMD Multimedia Extension define C/C++ language data types for vectors, as described in *Section 2.6* on page 59.

2.5.3 Addressing Modes

The PPE supports not only basic load and store operations but also load and store vector left or right indexed forms. All Vector/SIMD Multimedia Extension load and store operations use the register + register indexed addressing mode, which forms the sum of the contents of an index GPR plus the contents of a base-address GPR. This addressing mode is very useful for accessing arrays.

In addition to the load and store operations, the Vector/SIMD Multimedia Extension instructions provide a powerful set of element-manipulation instructions—for example, permute (similar to the SPEs' shuffle, *Figure 2-6* on page 56), rotate, and shift—to manipulate vector elements into the desired alignment and arrangement after the vectors have been loaded into vector registers.

2.5.4 Instruction Types

Most Vector/SIMD Multimedia Extension instructions have three or four 128-bit vector operands—two or three source operands and one result. Also, most instructions are SIMD in nature. The instructions have been chosen for their utility in digital signal processing (DSP) algorithms, including 3D graphics.

The Vector/SIMD Multimedia Extension instructions include the following types:

- *Vector Fixed-Point Instructions*—Vector arithmetic (including saturating arithmetic), compare, logical, rotate, and shift instructions. They operate on byte, halfword, and word vector elements.
- *Vector Floating-Point Instructions*—Floating-point, multiply/add, rounding and conversion, compare, and estimate instructions. They operate on single-precision floating-point vector elements.

- *Vector Load and Store Instructions*—Basic fixed-point and floating-point load and store instructions. No update forms of the load and store instruction are provided. They operate on 128-bit vectors.
- *Vector Permutation and Formatting Instructions*—Vector pack, unpack, merge, splat, permute, select, and shift instructions.
- *Processor Control Instructions*—Instructions that read and write the vector status and control register (VSCR).
- *Memory Control Instructions*—Instructions for managing caches (user-level and supervisor-level). These instructions are no-ops on the CBE processor.

2.5.5 Instructions

Table 2-3 summarizes the Vector/SIMD Multimedia Extension instructions, showing their pipeline, latency, and throughput. The Number of Instructions columns includes instructions versions that are recording (those that not only perform the stated function but also set bits in the Condition Register to reflect characteristic of the result of execution). A detailed list of the instructions, with their pipeline, latency, and throughput, is given in *Table A-8* on page 738. For a complete description of the instructions, see *PowerPC Microprocessor Family: Vector/SIMD Multimedia Extension Technolgy Programming Environments Manual*.

Table 2-3. Vector/SIMD Multimedia Extension Instruction Summary

Description	Number of Instructions	Latency[1]	Throughput (IPC)	Execution Unit[2]
Load Vector	10	2	1	VXU load, LSU
Move to or from VSCR	2	N/A	N/A	simple
Store Vector	9	N/A	1	VXU store, LSU
Vector Arithmetic	27	4	1	simple
Vector Compare	40	4	1	simple
Vector Logical	5	4	1	simple
Vector Merge, Replicate (splat), Permute	13	4	1	permute
Vector Multiply, Multiply-Add	22	9	1	complex
Vector Pack or Unpack	15	4	1	permute
Vector Rotate or Shift	14	4	1	simple
Vector Round Vector Convert	8	12	1	float
Vector Select	1	4	1	simple
Vector Shift	3	4	1	permute
Vector Single-Precision Floating-Point Arithmetic	4	12	1	float
Vector Single-Precision Floating-Point Log-2 Estimate	2	12	1	float
Vector Single-Precision Floating-Point Reciprocal Estimate	2	14	1	estimate

1. Some fields are labelled “N/A” to indicate that a more elaborate description is required (and is best understood by taking a broader view of the machine as a whole),

2. The Vector/SIMD Multimedia Extension and FPU instructions are categorized into two groups. The first group, VSU type 1, includes Vector/SIMD Multimedia Extension simple, Vector/SIMD Multimedia Extension complex, Vector/SIMD Multimedia Extension float, and FPU arithmetic instructions. The second group, VSU type 2, include Vector/SIMD Multimedia Extension load, Vector/SIMD Multimedia Extension store, Vector/SIMD Multimedia Extension permute, FPU load, and FPU store instructions

2.5.6 Graphics Rounding Mode

The CBE processor implements a graphics rounding mode that allows programs written with Vector/SIMD Multimedia Extension instructions to produce floating-point results that are equivalent in precision to those written in the SPU instruction set. In this Vector/SIMD Multimedia Extension mode, as in the SPU environment, the default rounding mode is r*ound-to-zero*, denormals are treated as zero, and there are no infinities or NaNs.

For details about the graphics rounding mode, see *Section A.3.3* on page 742.

2.6 Vector/SIMD Multimedia Extension C/C++ Language Intrinsics

Both the Vector/SIMD Multimedia Extension and SPU instructions are supported by C/C++ language extensions that define a set of vector data types and vector intrinsics. The intrinsics are essentially inline assembly-language instructions in the form of C-language function calls. The intrinsics provide explicit control of the underlying assembly instructions without directly managing registers and scheduling instructions, as assembly-language programming requires. A compiler that supports the C/C++ language extensions for the Vector/SIMD Multimedia Extension instructions will emit code optimized for the Vector/SIMD Multimedia Extension architecture.

Section A.4 on page 744 and the *AltiVec Technology Programming Interface Manual* provides details about the intrinsics, including a complete list of the intrinsics. Although the intrinsics provided by the PPE and SPE instruction sets are similar in function, their naming conventions and function-call forms are different. The Vector/SIMD Multimedia Extension intrinsics are summarized in this section. The SPE intrinsics are summarized in *Section 3.4* on page 76.

2.6.1 Vector Data Types

The vector registers are 128 bits and can contain:

- Sixteen 8-bit values, signed or unsigned.
- Eight 16-bit values, signed or unsigned.
- Four 32-bit values, signed or unsigned.
- Four single-precision IEEE-754 floating-point values.

The Vector/SIMD Multimedia Extension C/C++ language extensions define a set of fundamental data types, called *vector types*. These are summarized in *Table A-10* on page 744.

Introducing fundamental vector data types permits a compiler to provide stronger type-checking and supports overloaded operations on vector types. Since the token, *vector*, is a keyword in the Vector/SIMD Multimedia Extension data types, it is recommended that the term not be used elsewhere in the program as, for example, a variable name.

2.6.2 Vector Literals

The Vector/SIMD Multimedia Extension C/C++ language extensions define a set of vector literal formats. These are summarized in *Table A-11* on page 745. These formats consist of a parenthesized vector type followed by a parenthesized set of constant expressions.

2.6.3 Intrinsics

The Vector/SIMD Multimedia Extension intrinsics are summarized in *Table A-13* on page 746.

2.6.3.1 *Classes*

The Vector/SIMD Multimedia Extension intrinsics map directly to one or more Vector/SIMD Multimedia Extension assembly-language instructions. The intrinsics are grouped in the following three classes:

- *Specific Intrinsics*—Intrinsics that have a one-to-one mapping with a single assembly-language instruction. The specific intrinsics are rarely used.
- *Generic Intrinsics*—Intrinsics that map to one or more assembly-language instructions as a function of the type of input parameters. In general, the generic class of intrinsics contains the most useful intrinsics.
- *Predicate Intrinsics*—Intrinsics that compare values and return an integer that may be used directly as a value or as a condition for branching.

The specific and generic intrinsics use the prefix, "`vec_`" in front of an assembly-language or operation mnemonic; predicate intrinsics use the prefixes "`vec_all`" and "`vec_any`". When complied, the intrinsics generate one or more Vector/SIMD Multimedia Extension assembly-language instructions.

Vector/SIMD Multimedia Extension intrinsics can be used anywhere in a C/C++ language program. There is no need for setup or to enter a special mode.

2.6.3.2 *Example*

The following code example contains two versions of a function that sums an input array of 16 byte values. In such an array-summing function, one could unroll the scalar code to slightly improve the performance or one could use the Vector/SIMD Multimedia Extension intrinsics to significantly improve its performance and eliminate the loop entirely.

The first version, below, performs 16 iterations of the loop. The second version uses Vector/SIMD Multimedia Extension intrinsics to eliminate the loop entirely.

```
// 16 iterations of a loop
int rolled_sum(unsigned char bytes[16])
{
    int i;
    int sum = 0;
    for (i = 0; i < 16; ++i)
    {
        sum += bytes[i];
    }
```

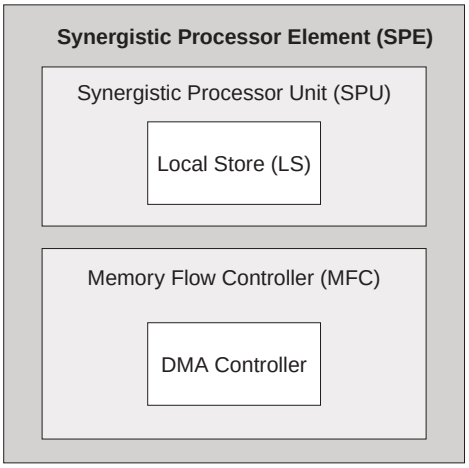
```
    return sum;
}


// Vectorized for Vector/SIMD Multimedia Extension
int vectorized_sum(unsigned char bytes[16])
{
  vector unsigned char vbytes;
  union {
    int i[4];
    vector signed int v;
  } sum;
  vector unsigned int zero = (vector unsigned int){0};

  // Perform a misaligned vector load of the 16 bytes.
  vbytes = vec_perm(vec_ld(0, bytes), vec_ld(16, bytes), vec_lvsl(0, bytes));

  // Sum the 16 bytes of the vector
  sum.v = vec_sums((vector signed int)vec_sum4s(vbytes, zero),
    (vector signed int)zero);

  // Extract the sum and return the result.
  return (sum.i[3]);
}
```

3. Synergistic Processor Elements

The eight Synergistic Processor Elements (SPEs) execute a new SIMD instruction set—the *Synergistic Processor Unit Instruction Set Architecture*. Each SPE is a 128-bit RISC processor specialized for data-rich, compute-intensive SIMD and scalar applications. It consists of two main units, the Synergistic Processor Unit (SPU) and the Memory Flow Controller (MFC), shown in *Figure 3-1*.

In this document, the term “SPU” refers to the instruction set or the unit that executes the instruction set, and the term “SPE” refers generally to functionality of any part of the SPE processor element, including the MFC.

Figure 3-1. SPE Block Diagram



The SPEs provide a deterministic operating environment. They do not have caches, so cache misses are not a factor in their performance. Pipeline-scheduling rules are simple, so it is easy to statically determine the performance of code and for software to generate high-quality, static schedules.

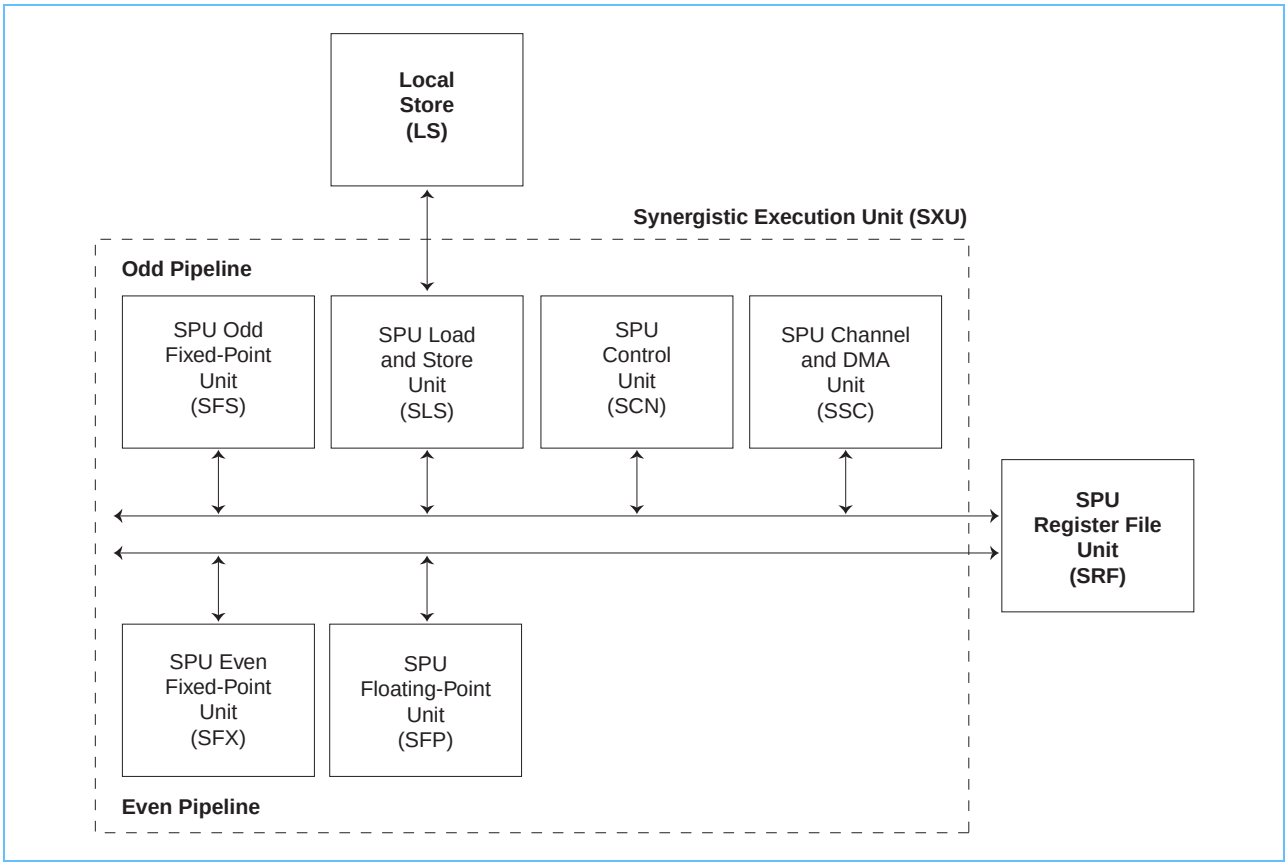
3.1 Synergistic Processor Unit

The SPU fetches instructions from its unified (instructions and data) 256-KB local store (LS), and it loads and stores data between its LS and its single register file for all data types, which has 128 registers, each 128 bits wide. The SPU has four execution units, a DMA interface, and a channel interface for communicating with its MFC, the PPE and other devices (including other SPEs).

Each SPU is an independent processor element with its own program counter, optimized to run SPU programs. The SPU fills its LS by requesting DMA transfers from its MFC, which implements the DMA transfers using its DMA controller. Then, the SPU fetches and executes instructions from its LS, and it loads and stores data to and from its LS.

The main SPU functional units are shown in *Figure 3-2*. These include the Synergistic Execution Unit (SXU), the LS, and the SPU Register File Unit (SRF). The SXU contains six execution units, which are described in *Section 3.1.3* on page 68.

Figure 3-2. SPU Functional Units



The SPU can issue and complete up to two instructions per cycle, one on each of the two (odd and even) execution pipelines. Whether an instruction goes to the odd or even pipeline depends on the instruction type. The instruction type is also related to the execution unit that performs the function, as summarized in *Section 3.3.2* on page 75.

3.1.1 Local Store

The Local Store (LS) is a 256-KB, ECC-protected, single-ported, noncaching memory. It stores all instructions and data used by the SPU. It supports one access per cycle from either SPE software or DMA transfers. SPU instruction prefetches are 128 bytes per cycle. SPU data-access bandwidth is 16 bytes per cycle, quadword aligned. DMA-access bandwidth is 128 bytes per cycle. DMA transfers perform a read-modify-write of LS for writes less than a quadword.

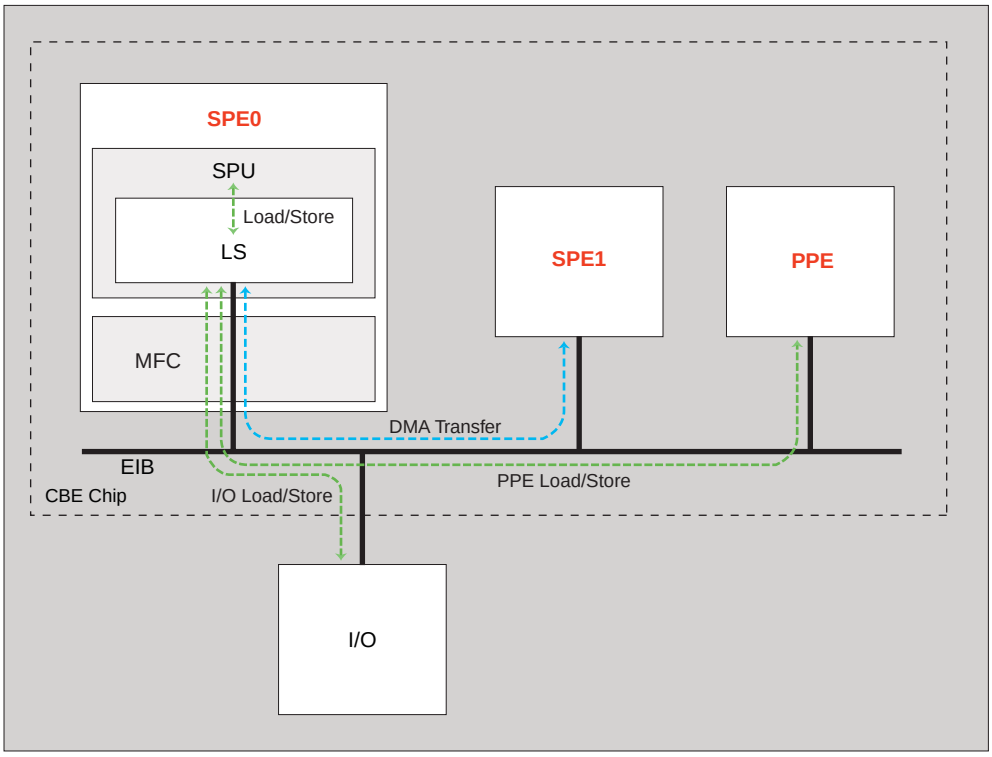
3.1.1.1 *Addressing and Address Aliasing*

The SPU accesses its LS with load and store instructions, and it performs no address translation for such accesses. Privileged software on the PPE can assign effective-address aliases to an LS. This enables the PPE and other SPEs to access the LS in the main-storage domain. The PPE performs such accesses with load and store instructions, without the need for DMA transfers.

However other SPEs must use DMA transfers to access the LS in the main-storage domain. When aliasing is set up by privileged software on the PPE, the SPE whose LS is being accessed performs address translation, as described in *Section 3.2 Memory Flow Controller* on page 70.

Figure 3-2 illustrates the methods by which an SPU, the PPE, other SPEs, and I/O devices access the SPU's associated LS, when the LS has been aliased to the main storage domain. See *Figure 1-2 Storage and Domains and Interfaces* on page 40 for an overview of the storage domains. For details about how the PPE does this aliasing, see *Section 4 Virtual Storage Environment* on page 79.

Figure 3-3. LS Access Methods



DMA	Direct Memory Access	MFC	Memory Flow Controller
EIB	Element Interconnect Bus	PPE	PowerPC Processor Element
LS	Local Store	SPE	Synergistic Processor Element

3.1.1.2 *Coherence and Synchronization*

DMA transfers between the LS and main storage are coherent in the system. A pointer to a data structure created on the PPE can be passed by the PPE, through the main-storage space, to an SPU, and the SPU can use this pointer to issue a DMA command to bring the data structure into its LS. Memory-mapped mailboxes or atomic MFC synchronization commands can be used for synchronization and mutual exclusion. See *Section 19* on page 505 and *Section 20* on page 551 for details.

3.1.1.3 *LS Access Priority*

Although the LS is shared between DMA transfers, load and store operations by the associated SPU, and instruction prefetches by the associated SPU, DMA operations are buffered and can only access the LS at most one of every eight cycles. Instruction prefetches deliver at least 17 instructions sequentially from the branch target. Thus, the impact of DMA operations on loads and stores and program-execution times is, by design, limited.

When there is competition for access to the LS by loads, stores, DMA reads, DMA writes, and instruction fetches, the SPU arbitrates access to the LS according the following priorities (highest priority first):

1. DMA reads and writes.
2. SPU loads and stores.
3. Instruction prefetch.

Table 3-1 summarizes the complete LS-arbitration priorities and transfer sizes. Memory-mapped I/O (MMIO) accesses and DMA transfers always have highest priority. These operations occupy, at most, one of every eight cycles (one of sixteen for DMA reads, and one of sixteen for DMA writes) to the LS. Thus, except for highly optimized code, the impact of DMA reads and writes on LS availability for loads, stores, and instruction fetches can be ignored.

Table 3-1. LS-Access Arbitration Priority and Transfer Size

Transaction	Transfer Size (Bytes)	Priority	Maximum LS Occupancy (SPU Cycle)	Access Path
MMIO[1]	≤ 16	1-Highest	1/8	Line Interface
DMA	≤ 128	1		
DMA List Transfer-Element Fetch	128	1	1/4	Quadword Interface
ECC Scrub[2]	16	2	1/10	
SPU Load or Store	16	3	1	
Hint Fetch[3]	128	3	1	Line Interface
Inline Fetch	128	4-Lowest	1/16 for inline code	

1. Access to LS contents by the PPE and other devices (including other SPEs) that can execute loads and stores in the main-storage space.
2. The DMA logic has an ECC-scrub state machine. It is initialized when an ECC error is detected during a load instruction or DMA read, and it accesses every quadword in LS to find and correct every ECC error.
3. See *Section 24.3.3 Branch Hints* on page 691.

After MMIO and DMA accesses, the next-highest priorities are given to ECC scrubbing. This is followed by data accesses by SPU load and store instructions and by hint fetches. The rationale for giving SPU loads and stores higher priority than inline instruction fetches (sequential instruction prefetch) is that loads and stores usually help a program's progress, whereas instruction fetches are often speculative. The SPE supports only 16-byte load and store operations that are 16-byte-aligned. The SPE uses a second instruction (byte shuffle) to place bytes in a different order if, for example, the program requires only a 4-byte quantity or a quantity with a different data alignment. To store data that is not aligned, use a read-modify-write operation.

The lowest priority for LS access is given to inline instruction fetches. Instruction fetches load 32 instructions per SPU request by accessing all banks of the LS simultaneously. Because the LS is single-ported, it is important that DMA and instruction-fetch activity transfer as much useful data as possible in each LS request.

3.1.2 Register File

The SPU's 128-entry, 128-bit register file stores all data types—integer, single- and double-precision floating-point, scalars, vectors, logicals, bytes, and others. It also stores return addresses, results of comparisons, and so forth. All computational instructions operate on registers—there are no computational instructions that modify storage. The large size of the register file, and its unified architecture, help mitigate the lack of a cache for local storage and achieve high performance without the use of expensive hardware techniques such as out-of-order processing or deep speculation.

The complete set of SPE problem-state (user) registers is shown in *Figure 3-4*. These registers include:

- *General-Purpose Registers (GPRs)*—All data types can be stored in the 128-bit GPRs, of which there are 128.
- *Floating-Point Status and Control Register (FPSCR)*—The processor updates the 128-bit FPSCR after every floating-point operation to record information about the result and any associated exceptions.

Figure 3-4. SPE Problem-State (User) Register Set

In addition to the problem-state registers shown in *Figure 3-4*, the SPU also supports privilege-state (supervisor) registers that are accessible by privileged software running on the PPE. The SPU does not have any Special-Purpose Registers (SPRs). The MFC, independently of the SPU, supports many MMIO registers and queues. For a summary, see *Section 3.2* on page 70. For details of all SPU and MFC registers, see the *Cell Broadband Engine Registers* specification.

3.1.3 Execution Units

The SPU supports dual-issue of instructions to its two execution pipelines, shown in *Figure 3-2* on page 64. The pipelines are referred to as even (pipeline 0) and odd (pipeline 1). The units execute the following types of operations:

- *SPU Odd Fixed-Point Unit (SFS)*—Executes quadword shift and rotates mask operations on bits, bytes, halfwords, and words, and shuffle operations on bytes.
- *SPU Even Fixed-Point Unit (SFX)*—Executes arithmetic instructions, logical instructions, word SIMD shifts and rotates, floating-point compares, and floating-point reciprocal and reciprocal square-root estimates.
- *SPU Floating-Point Unit (SFP)*—Executes single- and double-precision floating-point instructions, integer multiplies and conversions, and byte operations. The SPU supports only 16-bit multiplies, so 32-bit multiplies are implemented in software using 16-bit multiplies. For details on floating-point operations, see *Section 3.1.4*.
- *SPU Load And Store Unit (SLS)*—Executes load and store instructions and load branch-target-buffer (BTB) instructions. It also handles DMA requests to the LS.
- *SPU Control Unit (SCN)*—Fetches and issues instructions to the two pipelines, executes branch instructions, arbitrates access to the LS and register file, and performs other control functions.
- *SPU Channel and DMA Unit (SSC)*—Enables communication, data transfer, and control into and out of the SPU. The functions of SSC, and those of the associated DMA controller in the MFC, are described in *Section 3.2* on page 70.

A summary of the SPU instructions is given in *Table 3-3* on page 75. The SPU issues all instructions in program order according to the pipeline assignment. Each instruction is part of a double-word-aligned instruction pair called a *fetch group*. For details about the rules for fetching and issuing instructions, see *Section B.1.3* on page 768.

3.1.4 Floating-Point Support

The SPU supports both single- and double-precision floating-point operations. Single-precision instructions are performed in 4-way SIMD fashion, fully pipelined, whereas double-precision instructions are partially pipelined. The data formats for single- and double-precision instructions are those defined by IEEE Standard 754, but the results calculated by single-precision instructions depart from the IEEE Standard 754 by placing emphasis on real-time graphics requirements that are typical of multimedia processing.

For single-precision operations, the range of normalized numbers is extended beyond the IEEE standard. The representable, positive, non-zero numbers range from Smin = 2^{-126} to Smax = $(2 - 2^{-23}) \times 2^{128}$. If the exact result overflows (that is, if it is larger in magnitude than Smax), the rounded result is set to Smax with the appropriate sign. If the exact result underflows (that is, if it is smaller in magnitude than Smin), the rounded result is forced to zero. A zero result is always a positive zero.

Single-precision floating-point operations implement IEEE 754 arithmetic with the following extensions and differences:

- Only one rounding mode is supported: *round towards zero*, also known as *truncation*.
- Denormal operands are treated as zero, and denormal results are forced to zero.

- Numbers with an exponent of all ones are interpreted as normalized numbers and not as infinity or not-a-number (NaN).

Double-precision operations do not support the IEEE precise trap (exception) mode. If a double-precision denormal or not-a-number (NaN) result does not conform to IEEE Standard 754, then the deviation is recorded in a sticky bit in the FPSCR register, which can be accessed using the **fscrrd** and **fscrwr** instructions or the **spu_mffpscr** and **spu_mtfpscr** intrinsics.

Double-precision instructions are performed as two double-precision operations in 2-way SIMD fashion. However, the SPU is capable of performing only one double-precision operation per cycle. Thus, the SPU executes double-precision instructions by breaking up the SIMD operands and executing the two operations in consecutive instruction slots in the pipeline. Although double-precision instructions have 13-clock-cycle latencies, only the final seven cycles are pipe-lined. No other instructions are dual-issued with double-precision instructions, and no instructions of any kind are issued for six cycles after a double-precision instruction is issued.

Table 3-2 summarizes the SPE floating-point support and compliance with the IEEE Standard 754.

Table 3-2. SPE Floating-Point Support (Page 1 of 2)

Features		Single-Precision Floating-Point	Double-Precision Floating-Point
Minimum Representable Non-zero Value (Smin) [1]	Binary	(001)([1.]000... 000) (Extended-Range Mode)	(0001)([0.]000...001)
	Decimal	1×2^{-126} 1.2×10^{-38} (Extended-Range Mode)	$2^{-52} \times 2^{-1022}$ 4.9×10^{-324}
Maximum Representable Non-zero Value (Smax)	Binary	(255)([1.]111... 111) (Extended-Range Mode)	(2046)([1.]111...111)
	Decimal	$(2 - 2^{-23}) \times 2^{128}$ 6.8×10^{38} (Extended-Range Mode)	$(2 - 2^{-52}) \times 2^{1024}$ 1.8×10^{308}
Data Format		IEEE 754 compliant	IEEE 754 compliant
Zero	Input	Positive: all bits ‘0’	IEEE 754 compliant
		Negative: sign bit is ‘1’	
	Output	Positive: all bits ‘0’	
Not-a-Number (NaN) Support[2]	Operands and Results	NaNs treated as normalized numbers	SNaN, QNaN
	Propagation		Not IEEE 754 compliant (flag set on QNaN result)
Infinity (Inf) Support		Inf treated as normalized numbers	IEEE 754 compliant
Denormal Support	Operands	Treated as ‘0’	Not IEEE 754 compliant (treated as ‘0’, sign preserved, DENORM flag set in FPSCR)
	Results		IEEE 754 compliant
Rounding Modes Supported		Round to nearest	IEEE 754 compliant

Table 3-2. SPE Floating-Point Support (Page 2 of 2)

Features		Single-Precision Floating-Point	Double-Precision Floating-Point
Exception Support		Trapping (including precise trap) is not supported. OVF, UNF, DBZ, DIFF exceptions set flags in Extended-Range Mode only.[3]	Trapping (including precise trap) is not supported. INX, OVF, UNF, INV, NAN,DENORM exceptions set flags.[4]
Operand Conversion	DP-to-SP Rounding[5]	IEEE 754 compliant (except denormal inputs are forced to ‘0’)	
	SP-to-DP		
Operand-Conversion Result for DP-to-SP Rounding	Normalized Maximum (Smax)	(254)([1.]111... 111) $(2 - 2^{-23}) \times 2^{127}$ 3.4×10^{38}	—
	Denormalized Minimum (Smin)	(001)([0.]000... 001) $2^{-23} \times 2^{-126}$ 1.4×10^{-45}	

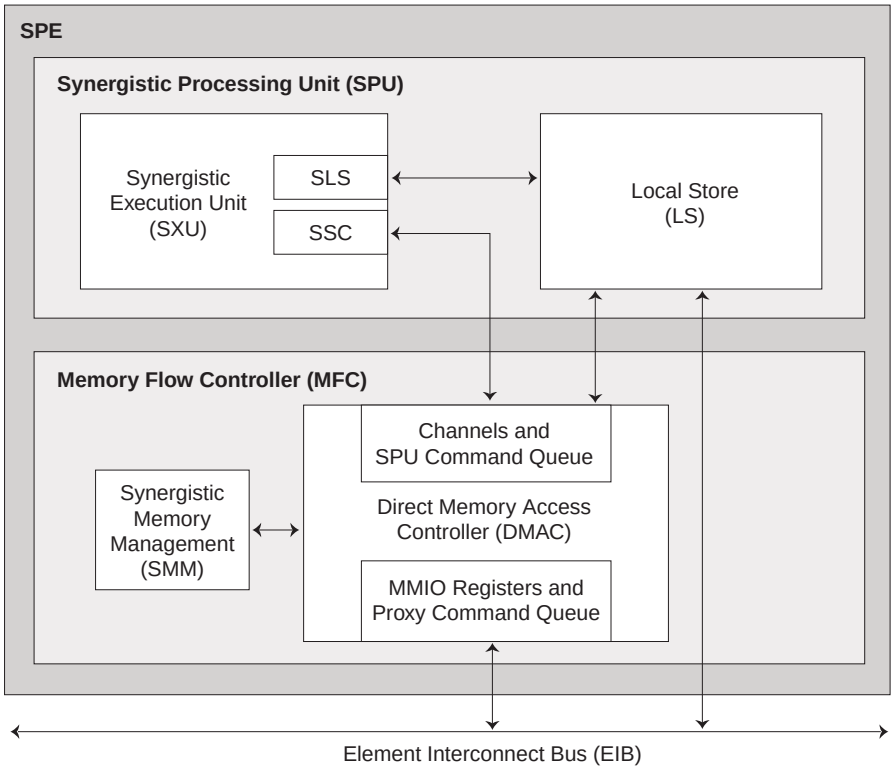
1. For Extended-Range Mode.
2. Signaling NaN (SNaN), Quiet NaN (QNaN).
3. For Extended-Range Mode: Overflow (OVF), Underflow (UNF), Divide-by-Zero (DBZ), IEEE noncompliant result (DIFF).
4. Inexact Result (INX), Overflow (OVF), Underflow (UNF), Invalid Operation (INV), Not Propagated NAN (NAN), Denormal Input Forced to Zero (DENORM)
5. Double-precision (DP), single-precision (SP).

3.2 Memory Flow Controller

Each SPU has its own Memory Flow Controller (MFC). The MFC serves as the SPU's interface, by means of the Element Interconnect Bus (EIB), to main-storage (*Figure 1-2* on page 40) and other processor elements and system devices. The MFC’s primary role is to interface its LS-storage domain with the main-storage domain. It does this by means of a DMA controller that moves instructions and data between its LS and main storage. The MFC also supports storage protection on the main-storage side of its DMA transfers, synchronization between main storage and the LS, and communication functions (such as mailbox and signal-notification messaging) with the PPE and other SPEs and devices.

Figure 3-5 shows a block diagram of the MFC.

Figure 3-5. MFC Block Diagram



MMIO	Memory-Mapped I/O Registers
SLS	SPU Load and Store Unit
SSC	SPU Channel and DMA Unit

Figure 3-5 shows the MFC's primary special-function interfaces to the SPU and EIB:

- *LS Read and Write Interface*. This links the LS with the EIB. It is used for DMA transfers (*Section 3.2.4.1* on page 73), atomic transfers (*Section 20.3* on page 587), and snoop requests from the EIB.
- *LS DMA List Element Read Interface*. This links the LS with the MFC's Channel and SPU Command Queue. It is used to implement efficient transfers of DMA lists (*Section 3.2.4.2* on page 73).
- *Channel Interface*. This links the SXU's SPU Channel and DMA Unit (SSC) with the MFC's Channel and SPU Command Queue. See *Section 3.2.1* on page 72 for an overview of channels.
- *EIB Command and Data Interfaces*. These link the MFC with the EIB. The data interface consists of a 128-bit data-in bus and a 128-bit data-out bus. As an EIB master, the MFC can send up to 16 outstanding MFC commands and it supports atomic requests. As an EIB slave, the MFC supports snoop requests (read and write) of the SPU's LS memory and the MFC's MMIO registers, and atomic requests.

3.2.1 Channels

SPE software communicates with main storage, the PPE, and other SPEs and devices through its channels, shown in *Figure 3-5* on page 71 and in *Figure 1-2* on page 40. Channels are unidirectional message-passing interfaces that support 32-bit messages and commands. Each SPE has its own set of channels. SPE software accesses channels with special channel-read and channel-write instructions that enqueue MFC commands (*Section 3.2.3*).

Software on the PPE and other SPEs and devices can gain access to most of an SPE's channel-interface functions by accessing associated problem-state (user) MMIO registers in the main-storage space.

For details about the channel facilities, see *Section 17 SPE Channel and Related MMIO Interface* on page 441.

3.2.2 Mailboxes and Signalling

The channel interface in each SPE supports two mailboxes for sending messages from the SPE to the PPE. One mailbox is provided for sending messages from the PPE to the SPE. The channel interface also supports two signal-notification channels (*signals*, for short) for inbound messages to the SPE.

The PPE is often used as an application controller, managing and distributing work to the SPEs. A large part of this task may involve loading main storage with data to be processed, then notifying an SPE by means of a mailbox or signal message. The SPE can also use its outbound mailboxes to inform the PPE that is has finished with a task.

3.2.3 MFC Commands and Command Queues

Software on the SPE, the PPE, and other SPEs and devices use MFC commands to initiate DMA transfers, query DMA status, perform MFC synchronization, perform interprocessor-communication via mailboxes and signal-notification, and so forth. The MFC commands that implement DMA transfers between the LS and main storage are called DMA commands.

The MFC maintains two separate command queues, shown in *Figure 3-5* on page 71—an SPU Command Queue for commands from the MFC's associated SPU, and a Proxy Command Queue for commands from the PPE and other SPEs and devices. The two queues are independent.

The MFC supports out-of-order execution of DMA commands. DMA commands can be tagged with one of 32 tag-group IDs. By doing so, software can determine the status of the entire group of commands. Software can, for example, use the tags to wait on the completion of queued commands in a group. Commands within a tag group can be synchronized with a fence or barrier option. The SPE can also use atomic-update commands that implement uninterrupted updates (read followed by write) of a 128-byte cache-line-size storage location.

For details about MFC commands and command queues, see *Section 19.2 MFC Commands* on page 506.

3.2.4 Direct Memory Access Controller

The MFC's DMA Controller (DMAC) implements DMA transfers of instructions and data between the SPU's LS and main storage. Programs running on the associated SPU, the PPE, or another SPE or device, can issue the DMA commands. (The PPE can also access an LS through the main-storage domain using load and store instructions.)

The MFC executes DMA commands autonomously, which allows the SPU to continue execution in parallel with the DMA transfers. Each DMAC can initiate up to 16 independent DMA transfers to or from its LS.

3.2.4.1 *DMA Transfers*

To initiate a DMA transfer, software on an SPE uses a channel instruction to write the transfer parameters to the MFC command-queue channels. Software on the PPE and other SPEs and devices performs the comparable function by writing transfer parameters to the main-storage addresses of the MMIO registers that correspond to those channels.

Six parameters need to be written, but a single C/C++ intrinsic can accomplish all of them. Special techniques, such as multibuffering, can be used to overlap DMA transfers with SPE computation.

DMA transfers are coherent with respect to main storage. Virtual-memory address-translation information is provided to each MFC by the PPE operating system. Attributes of system storage (address translation and protection) are governed by the page and segment tables of the *PowerPC Architecture*. Although privileged software on the PPE can map LS addresses and certain MFC resources to the main-storage address space, enabling the PPE or other SPUs in the system to access these resources, this aliased memory is not coherent in the system.

For details about the DMA transfers, including code examples, see *Section 19.2.1 DMA Commands* on page 508, *Section 19.3 PPE-Initiated DMA Transfers* on page 514, and *Section 19.4 SPE-Initiated DMA Transfers* on page 521. For examples of using double-buffering techniques to overlap DMA transfers with computation on an SPU, see *Section 24.1.2* on page 682.

3.2.4.2 *DMA List Transfers*

A DMA list is a sequence of *list elements* that, together with an initiating DMA list command, specifies a sequence of DMA transfers between a single area of LS and possibly discontinuous areas in main storage. Such lists are stored in an SPE's LS, and the sequence of transfers is initiated with a DMA list command. DMA list commands can only be issued by programs running on the associated SPE, but the PPE or other devices (including other SPEs) can create and store the lists in an SPE's LS. DMA lists can be used to implement scatter-gather functions between main storage and the LS.

Each list element in the list contains a transfer size, the low half of an effective address, and a stall-and-notify bit which can be used to suspend list execution after transferring a list element whose stall-and-notify bit is set. Each DMA transfer specified in a list can transfer up to 16 KB of data, and the list can have up to 2,048 (2 K) list elements.

For details about the DMA list transfers, including examples, see *Section 19.2.2 DMA List Commands* on page 509 and *Section 19.4.4* on page 527.

3.2.5 Synergistic Memory Management Unit

The MFC's Synergistic Memory Management (SMM) unit, shown in *Figure 3-5* on page 71, provides the address-translation and protection features defined by *PowerPC Architecture* for accesses to main storage. The SMM does this based on address-translation information provided to it by privileged PPE software.

Figure 3-3 on page 65 illustrates the manner in which the PPE, other SPEs, and I/O devices access the SPU's associated LS when the LS has been aliased to main storage. The SMM supports the address translation for such accesses. The SMM also supports address translations for DMA accesses by the associated SPU to main storage.

The SMM has an 8-entry Segment Lookaside Buffer (SLB) and a 256-entry Translation Lookaside Buffer (TLB) that supports 4-KB, 64-KB, 1-MB, and 16-MB page sizes. Neither the SMM nor the SPU has direct access to system control facilities, such as page-table entries. For details about address translation, see *Section 4 Virtual Storage Environment* on page 79.

The following sections introduce the SPU instruction set and its C/C++ intrinsics. These instructions and intrinsics, together with the MFC's own command set (described in *Section 19.2* on page 506), are used to control DMA transfers and other essential aspects of the MFC.

3.3 SPU Instruction Set

The SPU supports an instruction set, as specified by the *Synergistic Processor Unit Instruction Set Architecture* (ISA), and a set of C/C++ intrinsics, as specified by the *SPU C/C++ Language Extensions*. Although most of the coding for the CBE processor will be in a high-level language like C or C++, an understanding the SPU instruction set adds considerably to a programmer's ability to produce efficient code. This is particularly true because most of the intrinsics have mnemonics that relate directly to the underlying assembly-language mnemonics.

The SPU ISA operates primarily on SIMD vector operands, both fixed- and floating-point, with support for some scalar operands. The PPE and the SPE both execute SIMD instructions, but the two processors execute different instruction sets, and programs for the PPE and SPEs are often compiled by different compilers. *Section 2.5.1* on page 55 introduces and illustrates the concept of SIMD operations.

3.3.1 Data Types

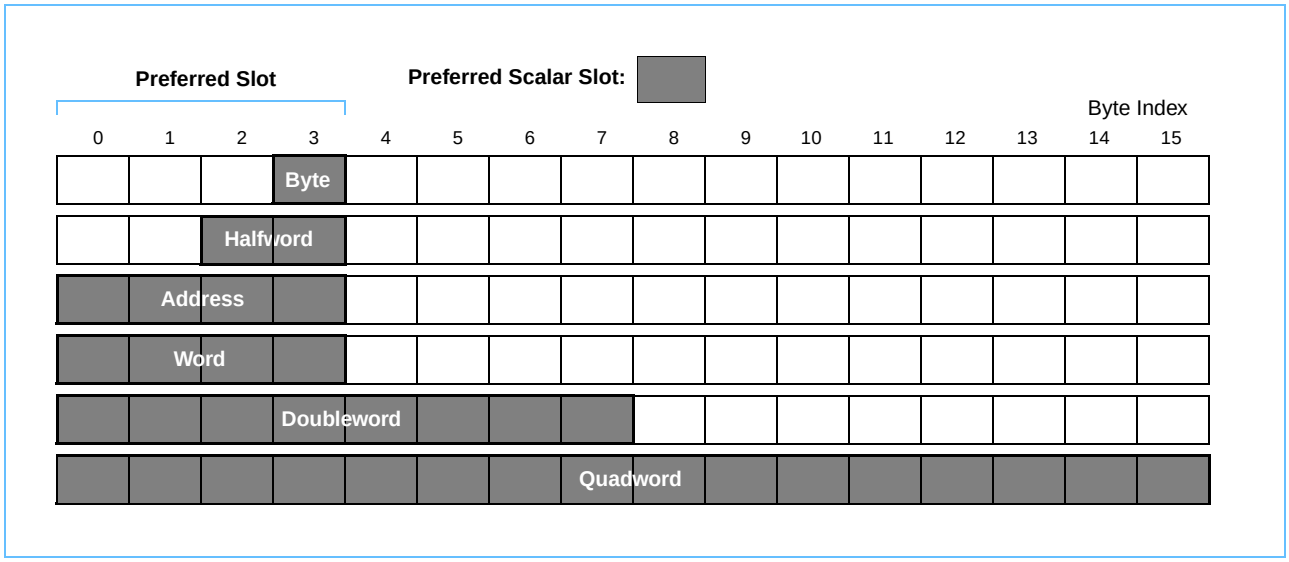
The SPU hardware supports the following data types:

- byte—8 bits
- halfword—16 bits
- word—32 bits
- doubleword—64 bits
- quadword—128 bits

These data types are indicated by shading in *Figure 3-6* on page 75. The left-most word (bytes 0, 1, 2, and 3) of a register is called the *preferred slot* in the *Synergistic Processor Unit Instruction Set Architecture*, although they are called the *preferred scalar slot* in the *SPU Application Binary Interface Specification*. When instructions use or produce scalar operands or addresses, the

values are in the preferred scalar slot. Since the SPU accesses its LS a quadword at a time, there is a set of store-assist instructions for insertion of bytes, halfwords, words, and double-words into a quadword for a subsequent store.

Figure 3-6. Register Layout of Data Types and Preferred Scalar Slot



3.3.2 Instructions

The SPU has two pipelines into which it can issue and complete up to two instructions per cycle, one in each of the pipelines. If two instructions with latency *n* and *m* are issued to the same pipeline, *n-m* cycles apart, the instructions would be retired in order, with the later one stalling for one cycle.

Whether an instruction goes to the even or odd pipeline depends on its instruction type. *Table 3-3* summarizes the instruction classes, the number of instructions in the class, and their latencies, pipeline assignments, and execution units. A detailed list of instructions is given in *Table B-1* on page 762. For details about the SPU instruction pipeline stages, see *Appendix B.1.3 Fetch and Issue Rules* on page 768.

Table 3-3. SPU Instruction Summary (Page 1 of 2)

Description	Instruction Class	Number of Instructions	Latency (clock cycles)	Pipeline	Execution Unit[1]
Load and Store	LS	12	6	Odd	SLS
Branch Hints	HB	4	15[2]	Odd	SLS
Branch Resolution[3]	BR	36	4	Odd	SCN
Channel Interface, Special Purpose Registers	CH	4	6	Odd	SSC
Shuffle	SH	35	4	Odd	SFS
No Operation (load)	LNOP	1	0	Odd	LNOP
No Operation (execute)	NOP	1	0	Even	NOP
Single-Precision Floating-Point	SP	6	6	Even	SFP

Table 3-3. SPU Instruction Summary (Page 2 of 2)

Description	Instruction Class	Number of Instructions	Latency (clock cycles)	Pipeline	Execution Unit[1]
Double-Precision Floating-Point	DP	10	13[4]	Even	SFP
Floating-Point Integer	FI	17	7	Even	SFP
Byte Operations	BO	3	4	Even	SFP
Simple Fixed-Point	FX	69	2	Even	SFX
Word Rotate and Shift	WS	17	4	Even	SFX

1. SLS: SPU Load and Store Unit, SCN, SPU Control Unit, SSC: SPU Channel and DMA Unit, SFS: SPU Odd Fixed-Point Unit, LNOP: Load Nop, NOP: Execute Nop, SFP: SPU Floating-Point Unit, SFX: SPU Fixed-Point Unit.
2. Includes the cycles required to load the hint-for branch buffer.
3. Inline or correctly hinted branches have zero-cycle delay. The mispredicted branch penalty is 18 to 19 cycles (depending on the address of the branch target), based on the program counter.
4. The last six cycles of a double-precision floating-point operation are instruction-issue stalls. No instructions of any kind are issued for six cycles after a double-precision floating-point instruction is issued.

3.4 SPU C/C++ Language Intrinsics

A set of C-language extensions, including data types and intrinsics, make the underlying SPU instruction set and hardware features conveniently available to C programmers. The intrinsics can be used in place of assembly-language code when writing in the C or C++ languages.

The intrinsics represent in-line assembly-language instructions in the form of C-language function calls that are built into a compiler. They provide the programmer with explicit control of the SPE SIMD instructions without directly managing registers. A compiler that supports these intrinsics will emit efficient code for the SPE architecture. The techniques used by compilers to generate efficient code should typically include:

- Register coloring.
- Instruction scheduling (dual-issue optimization).
- Data loads and stores.
- Loop blocking, fusion, unrolling.
- Correct up-stream placement of branch hints.
- Literal vector construction.

For example, a C or C++ compiler should generate a floating-point add instruction (**fa rt,ra,rb**) for the SPU intrinsic **`t = spu_add(a,b)`**, assuming **`t`**, **`a`**, and **`b`** are vector float variables. The system header file, `spu_intrinsics.h`, available from IBM, defines the SPU language extensions. The PPE and SPU instruction sets have similar, but distinct, intrinsics.

3.4.1 Vector Data Types

The C/C++ language extensions' vector data types are described in *Appendix B.2.1* on page 772.

3.4.2 Vector Literals

The C/C++ language extensions' vector literals are described in *Appendix B.2.2* on page 774.

3.4.3 Intrinsics

The C/C++ language extension intrinsics are grouped into the following three classes:

- *Specific Intrinsics*—Intrinsics that have a one-to-one mapping with a single assembly-language instruction. Programmers rarely need the specific intrinsics for implementing inline assembly code because most instructions are accessible through generic intrinsics.
- *Generic Intrinsics and Built-Ins*—Intrinsics that map to one of several assembly-language instructions or instruction sequences, depending on the type of operands.
- *Composite Intrinsics*—Convenience functions that map to assembly-language instruction sequences. A composite intrinsic can be expressed as a sequence of generic intrinsics.

Intrinsics are not provided for all assembly-language instructions. Some assembly-language instructions (for example, branches, branch hints, and interrupt return) are naturally accessible through the C/C++ language semantics.

Section B.2.3 on page 776 lists the most useful intrinsics. The intrinsics are defined fully in the *SPU C/C++ Language Extensions* document.

4. Virtual Storage Environment

4.1 Introduction

The CBE processor implements a virtual-storage environment that is compatible with that defined by the *PowerPC Operating Environment Architecture, Book III*. When virtual addressing is enabled, the combination of CBE-processor hardware and operating-system software supports a virtual-address space that is larger than either the effective-address (main-storage) space referenced by programs or the real-address space supported by the CBE processor.

Programs access memory by means of effective addresses. Programs on the PPE do this by executing instructions. Programs on an SPE do it by generating MFC DMA commands, which contain pairs of addresses—a Local Store (LS) address and an effective address (EA). The real-address (RA) space is the set of all addressable bytes in physical memory and on devices whose physical addresses have been mapped to the RA space—such as an SPE's on-chip LS or an I/O device's off-chip MMIO register or queue. The EA space is called the main-storage space and is illustrated in *Figure 1-2* on page 40.

The operating system can enable virtual-address translation (also called *relocation*) independently for instructions and data. When it does so, the address-translation mechanism allows each program running on the PPE or addressing main storage by means of an SPE's DMA-transfer facility to access up to 2^{64} bytes of EA space. Each program's EA space is a subset of the larger 2^{65} bytes of virtual-address (VA) space. The memory-management hardware in the PPE and in the MFC of each SPE translates EAs to VAs and then to RAs, to access 2^{42} bytes of RA space.

EAs are translated to VAs by a *segmentation* mechanism. *Segments* are protected, non-overlapping areas of virtual memory that contain 256 MB of contiguous addresses. VAs are translated to RAs by a *paging* mechanism. *Pages* are protected, non-overlapping, relocatable areas of real memory that contain between 4 KB and 16 MB of contiguous addresses.

Because the physical memory available in a system is often not large enough to map all virtual pages used by all currently executing programs, the operating system can transfer pages from disk storage to physical memory as demanded by the programs—a technique called *demand-paging*. The PPE performs address translation and demand-paging dynamically, during the execution of instructions. The operating system can revise the mapping information to relocate programs on the fly, during each demand-paged transfer from disk into memory. The PPE operating system manages address-mapping by setting up segment-buffer entries and page-table entries used by the hardware to locate addresses in the virtual-addressing environment. The PPE operating system also provides each MFC with the segment-buffer and page-table entries needed by the MFC's associated SPE for its DMA transfers to and from main storage.

In *PowerPC Architecture* paging, page-table entries perform an *inverted* mapping—from the small physical-address space of installed memory into the large (2^{65} bytes) virtual-address space. In contrast, conventional methods of paging map from a large virtual-address space to a small physical-address space. The inverted method of paging uses an *inverted page table,* which can be much smaller than conventional page tables. Whereas the size of conventional page tables is commensurate with the size of the virtual-address space allocated by the operating system, the size of a *PowerPC Architecture* inverted page table is commensurate with the size of installed memory. Lookup time for the inverted page table is reduced by using a hash algorithm.

In addition to supporting demand-paging, the virtual-address mechanism allows the operating system to impose restrictions on access to each segment and page[4], such as privilege-level and access-type restrictions, and that impose attributes (such as cacheability and memory coherence) on the storage being accessed.

If the operating system does not enable virtual addressing, the system operates in *real addressing mode*. In this mode (also called *real mode*), the EAs generated by programs are used as RAs[5], there is no page table, and there is no demand-paging. The only addressable memory is the physically installed memory.

This chapter provides an overview of, and several details about, the CBE processor's implementation of its virtual-memory environment and the means and techniques for loading and maintaining the related hardware resources. For more detail, including that related to the *PowerPC Architecture* in general, see the *PowerPC Architecture* books, the *PowerPC Microprocessor Family: The Programming Environments for 64- and 32-Bit Microprocessors*, and the *Cell Broadband Engine Architecture*.

4.2 PPE Memory Management

The PPE's Memory Management Unit (MMU) controls access by the PPE to main storage, including the manner in which EAs are translated to VAs and RAs.

Address translation is managed in the MMU by the following hardware units:

- *Effective-to-Real-Address Translation (ERAT) Buffers*—Two 64-entry caches, one for instructions and one for data. Each ERAT entry contains recent EA-to-RA translations for a 4-KB block of main storage, even if this block of storage is translated using a large page (*Section 4.2.6.3* on page 88).
- *Segment Lookaside Buffer (SLB)*—Two unified (instruction and data), 64-entry caches, one per PPE thread, that provide EA-to-VA translations. The PPE supports up to 2^{37} segments of 256 MB each. Segments are protected areas of virtual memory.
- *Page Table*—A page table is a hardware-accessed data structure in main storage that is maintained by the operating system. Page-table entries (PTEs) provide VA-to-RA translations. Pages are protected areas of real memory. There is one page table per logical partition (see *Section 11 Logical Partitions and a Hypervisor* on page 327).
- *Translation Lookaside Buffer (TLB)*—The TLB is a unified, 1024-entry cache that stores recently accessed PTEs. A page table in memory is only accessed if a translation cannot be found in the TLB (called a *TLB miss)*.

4. See *Section 4.2.5.1* on page 85 for segment protections and *Section 4.2.6.2* on page 87 for page protections.
5. For the PPE, but not the SPEs, RAs may be offset by a base address in real mode, as described in *Section 4.2.8* on page 99.

4.2.1 Memory Management Unit

Table 4-1 summarizes the features of the PPE's Memory Management Unit (MMU).

Table 4-1. PPE Memory Management Unit Features

Parameter		PPE Value or Feature
Addresses	Effective Address (EA) Size	64 bits.
	Virtual Address (VA) Size	65 bits[1]. This value is designated "n" in the *PowerPC Architecture*.
	Real Address (RA) Size	42 bits[2]. This value is designated "m" in the *PowerPC Architecture*.
ERAT	D-ERAT Entries	64, shared by both threads (2 way x 32).
	I-ERAT Entries	64, shared by both threads (2 way x 32).
SLB	SLB Entries	64 per thread.
	Segment Size	256 MB.
	Number of Segments	2^{37}.
	Segment Protection	Segments selectable as execute or no-execute.
	SLB Maintenance	Special instructions for maintaining the SLB.
Page Table	Page Sizes	4 KB, plus two of the following large-page sizes (p)[3]: • 64 KB (p = 16). • 1 MB (p = 20). • 16 MB (p = 24).
	Number of Page Tables	• With hypervisor[4]: one page table per logical partition. • Without hypervisor: one page table.
	Page Table Size (number of pages)	Determined by SDR1 register.
	Table Structure	Hashed page table(s) in memory. Inverted mapping, from small real-address space to large virtual-address space.
	Page Protection	Page Protection (PP) bits selectable as problem (user) or privileged (supervisor) state, and read-write or read-only.
	Page History	Referenced (R) and Changed (C) bits maintained.
TLB	TLB Entries	1024, shared by both threads.
	TLB Maintenance	• Hardware- or software-managed TLB update. • Special instructions for maintaining TLB. • Pseudo-LRU replacement policy. • Resource management table (RMT) for TLB replacement.

1. High-order bits above 65 bits in the 80-bit virtual address (VA[0:14]) are not implemented. The hardware always treats these bits as '0'. Software must not set these bits to any other value than '0' or the results are undefined in the PPE.
2. High-order bits above 42 bits in the 64-bit real address (RA[2:21]) are not implemented. The hardware always treats these bits as '0'. Software must not set these bits to any other value than '0' or the results are undefined in the PPE.The two most-significant bits (RA[0:1]) are architecturally reserved and must be '0'.
3. The value "p" is a power-of-2 variable in the *PowerPC Architecture* representing the size of a page.
4. See *Section 11 Logical Partitions and a Hypervisor* on page 327.

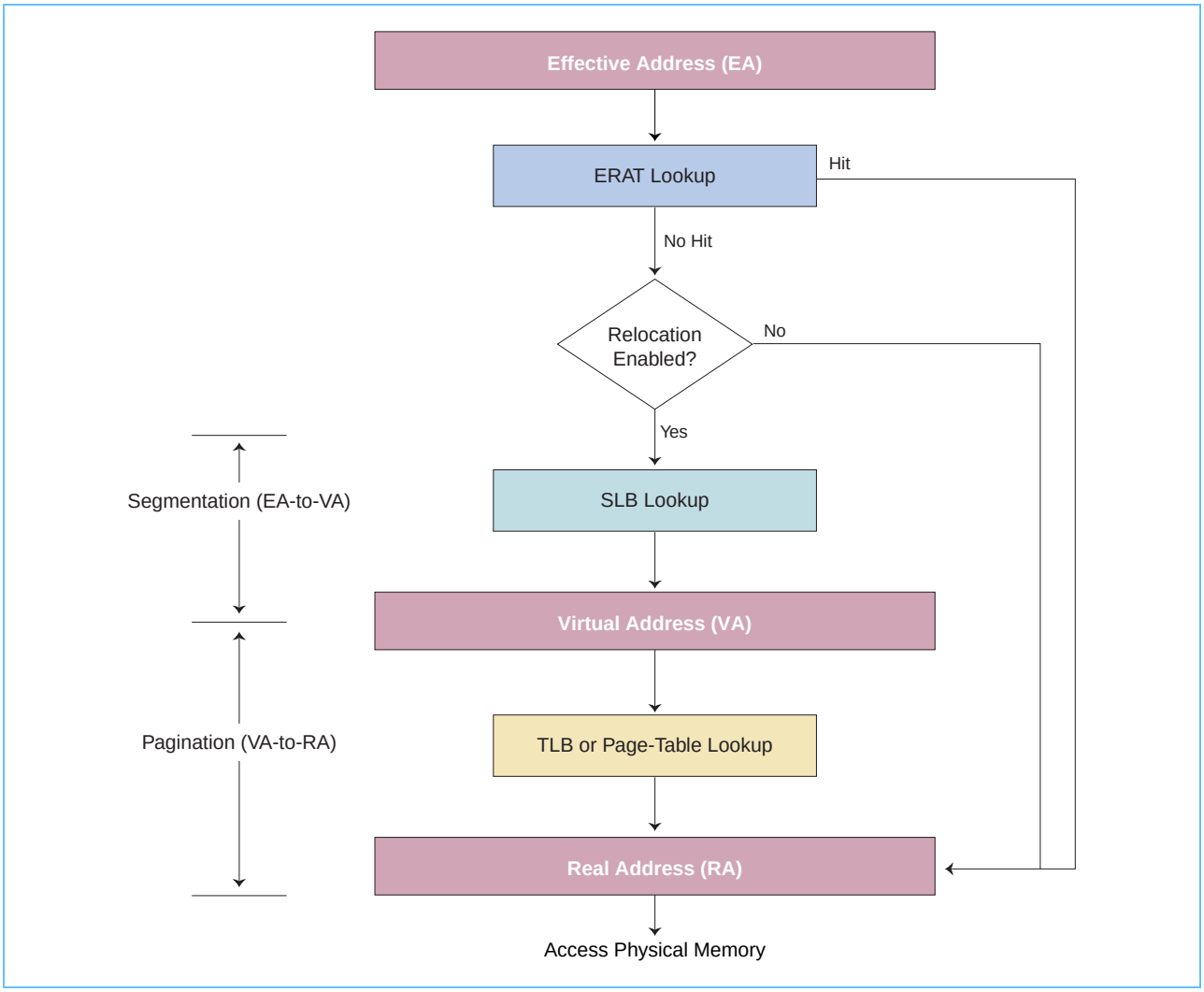
4.2.2 Address-Translation Sequence

Figure 4-1 gives an overview of the address-translation sequence. Two basic steps are used to convert an EA to an RA:

1. *Segmentation*—Convert the EA to a VA, using the SLB.
2. *Pagination*—Convert the VA to an RA, using a page table.

The SLB (*Section 4.2.5.1* on page 85) is used for the first conversion. The TLB (*Section 4.2.7* on page 92) caches recently used page-table entries and is used to enhances the performance of the second conversion. All EAs are first looked up in one of the two ERAT arrays. This ERAT lookup takes place regardless of whether the PPE is operating in virtual or real mode. If the address is not found in the ERAT, and address relocation is enabled, the address is sent to the segmentation and paging functions. If relocation is disabled, the real-mode translation facility is used.

Figure 4-1. PPE Address-Translation Overview



4.2.3 Enabling Address Translation

Virtual-address translation is enabled in the PPE by setting either or both of the Relocation bits in the Machine State Register (MSR) to ‘1’:

- Instruction Relocation (MSR[IR]).
- Data Relocation (MSR[DR]).

If either MSR[IR] = ‘0’ or MSR[DR] = ‘0’, PPE accesses to instructions or data (respectively) in main storage operate in real mode, as described in *Section 4.2.8 Real Addressing Mode* on page 99.

The following sections describe MMU address translation. Additional details are given in the Privileged-Mode Environment section of the *Cell Broadband Engine Architecture*, *Cell Broadband Engine Registers*, and the *PowerPC Operating Environment Architecture, Book III*.

4.2.4 Effective-to-Real-Address Translation

Recently used EA-to-RA translations are stored in the Effective-to-Real-Address Translation (ERAT) arrays. There are two such arrays—the Instruction ERAT (I-ERAT) array and the Data ERAT (D-ERAT).

4.2.4.1 *ERAT Functions*

The I-ERAT and D-ERAT are both 64-entry, 2-way set-associative arrays. They are not part of the normal *PowerPC Architecture* MMU. They provide faster address-translation information than the MMU. All main-storage accesses, regardless of the MSR[IR, DR] settings, pass through one of the two ERATs.

Each ERAT entry holds the EA-to-RA translation for an aligned 4-KB area of memory. When using a 4-KB page size, each ERAT entry holds the information for exactly one page. When using large pages, each ERAT entry contains a 4-KB section of the page, meaning that large pages can occupy several ERAT entries. All EA-to-RA mappings are kept in the ERAT including both real-mode and virtual-mode addresses (that is, addresses accessed with MSR[IR] equal to ‘0’ or ‘1’). The ERATs identify each translation entry with some combination of the MSR[SF, IR, DR, PR, and HV] bits, depending on whether the entry is in the I-ERAT or D-ERAT. This allows the ERATs to distinguish between translations that are valid for the various modes of operation.

The ERATs are shared by both threads, but all entries are identified by the thread ID that created the entry. Each thread maintains its own entries in the ERATs and cannot use the entries created for the other thread.

On instruction fetches, the least-significant bits of the EA are used to simultaneously index into the L1 ICache, the L1 ICache directory, the I-ERAT, and the Branch History Table (BHT); see *Section 6.1 PPE Caches* on page 133 for descriptions of the caches and the BHT. Several conditions need to be satisfied to consider the fetch a hit:

- The EA of the instruction must match the EA contained in the I-ERAT entry being indexed, and the I-ERAT entry must also be valid.
- The Instruction Relocate (IR) and Hypervisor State (HV) bits from the MSR must match the corresponding bits in the I-ERAT, which were set at the time the I-ERAT entry was loaded.
- The thread performing the fetch must match the thread ID from the I-ERAT entry.

- The RA from the L1 ICache directory must match the RA provided by the I-ERAT.
- The page-protection bits must allow the access to occur.

If all of these conditions are met, and if no L1 ICache parity errors have occurred, the fetch is a hit. If an instruction misses in the I-ERAT, it is held in an ERAT miss queue (there is one miss queue for instructions and another for data) while a request is sent to the MMU for translation. This invokes an 11-cycle penalty in order to translate the address in the MMU. The instruction following the ERAT miss is flushed, refetched, and held at dispatch while the other thread is given all of the dispatch slots. If this translation hits in the TLB, the ERAT entry is reloaded and the fetch is attempted again.

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

4.2.4.2 *ERAT Maintenance*

One MMIO register field affects use of the ERATs: the ERAT Cache Inhibit (`dis_force_ci`) bit in the Hardware Implementation Register 4 (`HID4[dis_force_ci]`). This bit should be set for normal use of the ERATs.

The replacement policy used by the ERATs is a simple 1-bit LRU policy. Each ERAT entry is set to the invalid state after Power-On Reset (POR). Normal ERAT operation is maintained by hardware. Any load or store that misses the appropriate ERAT causes all subsequent instructions to flush while that ERAT attempts to reload the translation from the MMU. However, in a hypervisor-controlled system, the ERAT entries are unique to each logical partition (LPAR) such that when a partition is switched by the hypervisor, the ERAT entries must be invalidated.

Each ERAT entry is obtained from a page-table search based on the contents of an SLB entry. To maintain consistency with the SLB, the following instructions cause all entries in an ERAT to be invalidated:

- **slbia** (invalidates entries belonging to the same thread only).
- **tlbie(l)**[6] to a large page only (invalidates all entries regardless of thread).

The **slbie** instruction causes a thread-based invalidation of the ERAT that is sensitive to the SLB Class (see *Section 4.2.5.1* on page 85). That is, it invalidates any entries that have a thread-ID and Class-bit match with the thread that issued and the segment Class indicated by the **slbie** instruction. In addition, the execution of **tlbie(l)** to a small page (L = ‘0’), or the detection of a snooped-**tlbie** operation from another processor element, causes an index-based invalidate to occur in the ERAT. All ERAT entries that have EA bits `[47:51]` matching the `RB[47:51]` bits provided by the instruction are invalidated.

When operating in real mode (*Section 4.2.8 Real Addressing Mode* on page 99), the following sequence of instructions must be used to clear the ERAT without affecting the TLB:

```
mtspr RMOR/HRMOR/LPCR/HID6
isync
tlbie L = 1, IS = 00, VPN(38:79-p) = non-mapped virtual address
sync
```

6. The notation, **tlbie(l)**, indicates either the **tlbie** or the **tlbiel** instruction.

This sequence should be executed at the end of all real-mode phases, before enabling virtual-address translation, because the ERAT is always referenced, be it real or virtual address.

4.2.5 Segmentation

Segmentation is implemented with the Segment Lookaside Buffers (SLBs). The SLBs translate EAs to VAs. The PPE contains two unified (instruction and data), 64-entry, fully associative SLBs, one per thread. Segments are protected areas of virtual memory. Each 256-MB segment can hold 64K pages of 4KB each (or fewer of larger page sizes). The pages that make up a segment can overlap, but they must do so completely, not partially. The operating system can concatenate several contiguous segments (segments referenced in contiguous segment registers) to create very large address spaces. The first byte in a segment is also the first byte in the first page of that segment, and the last byte in the segment is the last byte in the last page.

The *small* or *flat* segmentation model is the simplest. It uses a single segment for all instructions, data, and stacks. In this model, virtual-address translation is, in effect, implemented entirely with paging. Segmentation gives the operating system flexibility in allocating protected address spaces to specialized areas of memory.

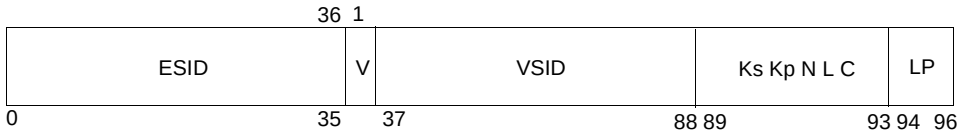
Multiple segments can be allocated for use by the instructions, private data, and stack of each process. They can also be allocated for shared libraries, shared memory-mapped I/O ports, and so forth. The operating system can also map file images from disk into shared segments, allowing applications to access file elements by moving a pointer or array index rather than by performing reads and writes through an I/O buffer. Two processes can thereby open the same file from different program-generated addresses and the accesses can map to the same area in the virtual address space. Access to segments can be protected by access-type and privilege attributes, so processes and data can be shared without interference from unauthorized programs.

4.2.5.1 *Segment Lookaside Buffer*

Figure 4-2 on page 85 shows the format for an SLB entry (SLBE). Each entry specifies the mapping between an Effective Segment ID (ESID) and a Virtual Segment ID (VSID), which are the most-significant bits of the EA and VA, respectively. The PPE supports up to 2^{37} segments, as specified in the bits 52:88 of the VSID field. The PPE supports a 65-bit virtual address, so bits [37:51] of an SLB Entry (`VSID[0:14]`) are always ‘0’.

When hardware searches the SLB, all entries in the SLB are tested for a VA match with the EA. If the SLB search fails, an Instruction Segment or Data Segment exception occurs, depending on whether the effective address is for an instruction fetch or for a data access.

Figure 4-2. SLB Entry Format



The other fields in the SLB entry include the Valid (V), Supervisor Key (Ks), Problem Key (Kp), No-execute (N), Large Page Indicator (L), Class (C), and Large Page Selector (LP) fields. See the *Cell Broadband Engine Architecture* for definitions of these fields. The PPE adds the Large Page (LP) field to the *PowerPC Architecture* SLB Entry (see *Section 4.2.6.3 Large Pages* on page 88). The LP field is shown in *Figure 4-2* to be 3 bits wide, as specified in the *Cell Broadband Engine Architecture*. However, the PPE implements only the low-order bit of the LP field. For this reason, other LP references in this document describe the LP field as a single bit.

4.2.5.2 *SLB Maintenance*

Software must ensure that the SLB contains at most one entry that translates a given EA (that is, a given ESID must be contained in no more than one SLB entry). Software maintains the SLB using the **slbmte**, **slbmfee**, **slbmfev**, **slbie**, and **slbia** instructions. All other optional *PowerPC Architecture* instructions (**mtsr**, **mtsrin**, **mfsr**, and **mfsrin**) provided for 32-bit operating systems and for changing the contents of the SLB are not implemented in the PPE.

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

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

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

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

For the effect these instructions have on the ERATs, see *Section 4.2.4 Effective-to-Real-Address Translation* on page 83.

Information derived from the SLB can be cached in the I-ERAT or the D-ERAT along with information from the TLB. As a result, many of the SLB-management instructions have effects on the ERATs as well as on the SLB itself. Because the SLB is managed by the operating system, it is possible that multiple entries may be incorrectly set up to provide translations for the same EA. This will result in undefined behavior and can result in a Checkstop.

4.2.5.3 *SLB After POR*

All SLB entries and TLB entries are invalidated after Power-On Reset (POR).

4.2.6 Paging

Paging is set up by the operating system and can be actively managed by hardware or software, as described in *Section 4.2.7.1 Enabling Hardware or Software TLB Management* on page 92. The operating system creates page-table entries (PTEs), and hardware or software loads recently accessed PTEs into the translation-lookaside buffer (TLB). A high percentage of paging accesses—typically in excess of 95%—hit in the TLB.

On systems that support disk peripherals, only recently accessed pages are kept in physical memory; the remaining pages are stored on disk. If a program attempts to access a page containing instructions or data that are not currently in physical memory, a *page fault* occurs and the operating system must load the page from disk and update the page table with the physical and virtual addresses of that page.

4.2.6.1 *Page Table*

The operating system must provide PTEs in the page table for all of real memory. There is one page table per logical partition (see *Section 11 Logical Partitions and a Hypervisor* on page 327), or one page table per system when logical partitions are not used. The operating system places PTEs in the page table by using a hashing algorithm. The Hashed Page Table (HTAB) is a variable-sized data structure. Each PTE in the HTAB is a 128-bit data structure that maps one Virtual Page Number (VPN) to its currently assigned Real Page Number (RPN), thus specifying a mapping between VA and RA. The PTE also provides protection inputs to the memory-protection and cache mechanisms.

Each process can have its own segmented address space in the virtual space, but there is only one page table that is shared globally by all processes in a logical partition. During context switches, segments are changed but the TLB is not necessarily purged and the page table remains unchanged. To change a segment, the operating system simply loads a new value into the SLB, which is faster than reloading the TLB. A hashing function applied to a VA provides an index into the page table. The entire page table needed to map, for example, 1 GB of real memory would be a minimum of 2MB in size (1 GB, times 8 bytes per PTE, divided by 4 KB per page). This architectural economy of paging overhead accounts, in part, for the superior performance of memory management in *PowerPC Architecture* processors, as compared with some other processor architectures.

4.2.6.2 *Page Table Entries*

Table 4-2 summarizes the implemented bits of a PTE. Software should set all unimplemented bits in the page table to ‘0’. *Table 4-2* views the PTE as a quadword when giving bit definitions. For more information, see the *PowerPC Operating Environment Architecture, Book III* and the *Cell Broadband Engine Architecture*.

Table 4-2. PTE Fields (Page 1 of 2)

Bit(s)	Field Name	Description
0:14	Reserved	Software must set these bits to ‘0’ or results are undefined.
15:56	AVPN	Abbreviated Virtual Page Number.
57:60	SW	Available for software use.
61	L	Large Page Indicator.

Table 4-2. PTE Fields (Page 2 of 2)

Bit(s)	Field Name	Description
62	H	Hash Function Identifier.
63	V	Valid.
64:85	Reserved	Software must set these bits to ‘0’ or results are undefined.
86:**115**	RPN	Real Page Number (but see bit 115, immediately below).
115	LP	Large Page Selector (last bit of RPN if L = ‘1’).
116:117	Reserved	Software must set these bits to ‘0’ or results are undefined.
118	AC	Address Compare bit.
119	R	Reference bit.
120	C	Change bit.
121	W	Write-through. Hardware always treats this as ‘0’. See *Table 4-4* on page 91.
122	I	Caching-inhibited bit.
123	M	Memory-Coherence. Hardware always treats this as ‘1’ if I ‘0’, or ‘0’ if I = ‘1’. See *Table 4-4* on page 91.
124	G	Guarded bit.
125	N	No Execute bit.
126:127	PP[0:1]	Page-Protection bits 1:2.

4.2.6.3 *Large Pages*

The page size can be 4 KB or any two of the three Large Page sizes: 64 KB, 1 MB, or 16 MB. Large pages reduce page-table size and increase TLB efficiency. They are useful in situations that may suffer from demand-paged thrashing, because they minimize the overhead of filling real memory if single large pages can be operated on for extended periods of time.

The selection between the two large page sizes is controlled by the lower bit of the Large Page Selector (`LP`) field of the PTE when the Large Page Mode (`L`) bit of the PPE Translation Lookaside Buffer Virtual-Page Number Register (`PPE_TLB_VPN[L]`) is set to ‘1’.

The address of the real page (that is, the Real Page Number, or RPN) is formed by concatenating the PTE’s Abbreviated RPN (`ARPN`) with the upper seven bits of the `LP` field and a ‘0’. Since the RPN must be on a natural page boundary, some bits of the `ARPN` and `LP` fields may be required to be ‘0’. If the `L` bit is not set in the `PPE_TLB_VPN` register, the RPN is formed by concatenating the `ARPN` with the whole `LP` field, and the page size is 4 KB.

4.2.6.4 *Page-Table Organization and Lookups*

The *Cell Broadband Engine Architecture* adds a bit to the *PowerPC Architecture* definition of the Logical Partition Control Register (`LPCR`)—the TLB Load (`TL`) bit—that specifies whether TLB misses are handled by hardware or software. In the hardware-managed TLB mode, when a virtual address is presented to the TLB and no matching TLB entry is found, the hardware initiates a reload of the TLB from the page table. In the software-managed TLB mode, a page fault is generated immediately following any TLB miss, and the MMU will not attempt to reload the TLB. For details on software accessibility and maintenance of the TLB, see *Section 4.2.7.4* on page 95.

Each logical partition can set LPCR[TL] differently. Changes to its value (see *Section 4.2.7.1* on page 92 and *Section 11.2.3* on page 332) must follow all the same synchronization rules as are required when changing the LPID. See *Section 20 Shared-Storage Synchronization* on page 551 and the *PowerPC Operating Environment Architecture, Book III* for the required synchronization instructions and sequences that must be used by the hypervisor. Failure to properly synchronize these changes can lead to inconsistent state and failure of the application or hypervisor.

In general, the MMU does not do speculative table-lookups (tablewalks). The only situation in which the MMU performs a speculative table-lookup is a caching-inhibited store that takes a Alignment interrupt. Otherwise, the MMU waits to perform the table-lookup until the instruction is past the point at which any exception can occur.

The PPE always performs page-table lookups and PTE writes in real mode (see *Section 4.2.8* on page 99). Because the PPE only supports a 42-bit RA, the following restrictions regarding the Storage Description Register 1 (SDR1) apply to hardware table-lookups:

- Maximum Hash Table Size (HTABSIZE) allowed in SDR1 is 24 (‘11000’).
- Maximum Hash Table Origin (HTABORG) allowed in SDR1 is x‘FFFFFF’.

The 16-byte PTEs are organized in memory as groups of eight entries, called PTE Groups (PTEGs), each one a full 128-byte cache line. A hardware table-lookup consists of searching a primary PTEG and then, if necessary, searching a secondary PTEG to find the correct PTE to be reloaded into the TLB. The L2 provides the PTEG data two 16-byte PTEs at a time, in a burst of 32 bytes. These 32 bytes are then run through a 2:1 multiplexer, producing 64 bytes—half of a primary or secondary PTEG. Searching of the primary or secondary PTEG is done as follows: first, the 64-byte even half (PTEs 0, 2, 4, and 6) is searched; if the PTE is not found here, the 64-byte odd half (PTEs 1, 3, 5, and 7) is searched. To accomplish two such searches, the data must be re-requested from the L2 for the odd half of the PTEG, and the multiplexer must be set to look at the appropriate half of the PTEG.

To summarize, hardware searches PTEGs in the following order:

1. Request the even primary PTEG entries.
2. Search PTE[0], PTE[2], PTE[4], and PTE[6].
3. Request the odd primary PTEG entries.
4. Search PTE[1], PTE[3], PTE[5], and PTE[7].
5. Repeat Steps 1 through 4 with the secondary PTE.
6. If no match occurs, raise a Data Storage exception.

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

4.2.6.5 *Page-Table Match Criteria*

The page-table match criteria are described in the Page Table Search section of *PowerPC Operating Environment Architecture, Book III*. The PPE also requires that PTE[LP] = SLBE[LP] whenever PTE[L] = ‘1’. In other words, the PTE page size must match the SLBE page size exactly for a page-table match to occur. If more than one match is found in the page table, then the matching entries must all have the same PTE[LP] value, or the results are undefined, as explained below.

To conserve bits in the PTE, the least-significant bit, bit 115, of the PTE[RPN] field doubles as the PTE[LP] bit. The PTE[LP] bit exists in the page table only if PTE[L] = '1'. When this is true, PTE[LP] is no longer part of the PTE[RPN] field and instead serves as the PTE[LP] bit. See *Section 4.2.6.6* on page 90 for details.

If multiple PTEs match yet differ in any field other than SW, H, AC, R, or C, then the results are undefined. The PPE will either return the first matching entry encountered or it will logically OR the multiple matching entries together before storing them in the TLB. This can lead to unpredictable address-translation results, and the system may Checkstop as a result.

The PPE does not implement PTE[0:15]. Software must zero these bits in all page table entries. The PPE will ignore these bits if set, and a match will occur as if PTE[0:15] = x'0000'.

4.2.6.6 *Decoding for Large Page Sizes*

The PPE supports two large page sizes concurrently in the system, in addition to the small 4-KB page size. Each page can be a different size, but each segment must consist of pages that are all of the same size (see *Section 4.2.6.5* on page 89).

Two large page sizes can be chosen from a selection of three sizes (64 KB, 1 MB, and 16 MB). To accomplish this, the PPE defines the LP field in the PTE to override the low-order bit of the RPN in the PTE and the low-order bit of the VPN in the **tlbie(l)** instruction if the large page (L) bit is set to '1' for the PTE or **tlbie(l)** instruction. Whenever L = '1', RPN[51], and VPN[67] are implicitly '0'. This is why the PPE can override these bits to be a large page selector (LP) whenever L equals '1'.

There are two possible values of the LP field ('0' and '1'), which allow two different large page sizes to be concurrently active in the system. To determine the actual page size, four bits are kept in HID6[LB], bits [16:19]. *Table 4-3* shows how the page size is determined given L, LP, and LB, where "x" means that any value is allowed.

Table 4-3. Decoding for Large Page Sizes

L	LP	LB[0:1]	LB[2:3]	Page Size
0	x	x	x	4 KB
1	0	11	x	Reserved
1	0	10	x	64 KB
1	0	01	x	1 MB
1	0	00	x	16 MB
1	1	x	11	Reserved
1	1	x	10	64 KB
1	1	x	01	1 MB
1	1	x	00	16 MB

Each logical partition can have a different LB value. Thus, support for large page sizes can be selected on a per-partition basis. However, the hypervisor is only permitted to change the LB field in HID6 if it is switching to or creating a partition. The partition is not allowed to dynamically change this value. The following sequence (or a similar one) must be issued in order to clean up translation after changing LB:

1. Issue **slbia** for thread 0.
2. Issue **slbia** for thread 1.
3. For (i = ‘0’; i < 256; i++)

   tlbiel is = 11 rb[44:51] = i; // invalidate the entire TLB by congruence class.
4. **sync** L = ‘0’.
5. **mtspr** `hid6`.
6. **sync** L = ‘0’.
7. **isync**.

Note: *The above procedure can impact performance negatively and should only be used when switching between partitions that use a different set of large page sizes.*

4.2.6.7 *WIMG-Bit Storage Control Attributes*

The PPE supports a *weakly consistent* shared-storage model, as described in *Section 20 Shared-Storage Synchronization* on page 551. Storage consistency between processing elements and I/O devices is controlled by privileged software using the following four storage-control attributes and their associated bits (called the *WIMG* bits):

- Write-through (W bit).
- Caching-inhibited (I bit).
- Memory coherency (M bit).
- Guarded (G bit).

All PPE instruction accesses and all PPE and SPE data accesses to main storage are performed under the control of these storage attributes. The attributes are maintained in each PTE. The W and I bits control how the processor element performing an access uses its own cache (in the case of the CBE processor, this applies to the PPE L1 and L2 caches and the SPE atomic caches). The M bit ensures that coherency is maintained for all copies of the addressed memory location. When an access requires coherency, the processor element performing the access informs the coherency mechanisms throughout the system of this requirement. The G bit prevents out-of-order loading and prefetching from the addressed location.

Like many PowerPC processors, the PPE supports only three WIMG-bit combinations: ‘0010’, ‘0100’, and ‘0101’. All other settings are implemented as shown in *Table 4-4*. In this table, a dash (–) represents either a ‘0’ or a ‘1’. If software writes an unimplemented WIMG value to a page-table or TLB entry, the W and M bits may be implicitly overwritten in that entry, to the value shown in *Table 4-4*, if an update to a Reference or Change bit occurs (see *Section 4.2.6.8* on page 92).

Table 4-4. Summary of Implemented WIMG Settings

WIMG Setting	PPE WIMG Implementation
– 0 – –	0 0 1 0
– 1 – 0	0 1 0 0
– 1 – 1	0 1 0 1

As *Table 4-4* shows, if a storage access is cacheable, then it is also write-back, memory-coherent, and nonguarded. See *Section 4.2.8.2* on page 100 for details about how the storage-control attributes are set in real mode. For further details about storage control attributes in real mode, including those for hypervisor accesses, see the Real Addressing Mode section of the *PowerPC Operating Environment Architecture, Book III*.

The storage attributes are relevant only when an EA is translated. The attributes are not saved along with data in the cache (for cacheable accesses), nor are they associated with subsequent accesses made by other processor elements. Memory-mapped I/O operations address main storage and are therefore subject to the same WIMG attributes as all other main-storage accesses.

4.2.6.8 *Updates to Reference and Change Bits*

If a hardware table-lookup is performed and a matching PTE entry is found with the Reference (R) bit set to ‘0’, the MMU performs a PTE store update, with the R bit set to ‘1’. Therefore, since all table-lookups are nonspeculative (with the exception of a caching-inhibited misaligned store[7]), the R bit is always updated for any page that has a load or store request, regardless of any Storage interrupts that might occur.

If a hardware table-lookup is the result of a store operation, a matching PTE is tested to see if the Change (C) bit is set to ‘0’. If so, the MMU performs a PTE store update, with the C bit set to ‘1’. The only case in which this is not true is a Storage exception. The MMU does not update the C bit if the storage access (load or store) results in a Storage exception, as described in the Storage Protection section of *PowerPC Operating Environment Architecture, Book III*. The PPE performs all store operations, including PTE updates, in-order.

A store operation to a page whose corresponding PTE’s C bit is ‘0’ in the TLB entry causes a page fault when in software-managed TLB mode (`LPCR[TL]` = ‘1’). To avoid this, software can set C = ‘1’ on the initial load of a new TLB entry, so that an additional page fault is not incurred during the subsequent store operation.

4.2.7 Translation Lookaside Buffer

The PPE’s Translation Lookaside Buffer (TLB) is a unified (combined both instruction and data), 1024-entry, 4-way set associative cache that stores recently accessed PTEs. It is shared by both PPE threads. A page table in memory is only accessed if a translation cannot be found in the TLB (called a TLB miss).

4.2.7.1 *Enabling Hardware or Software TLB Management*

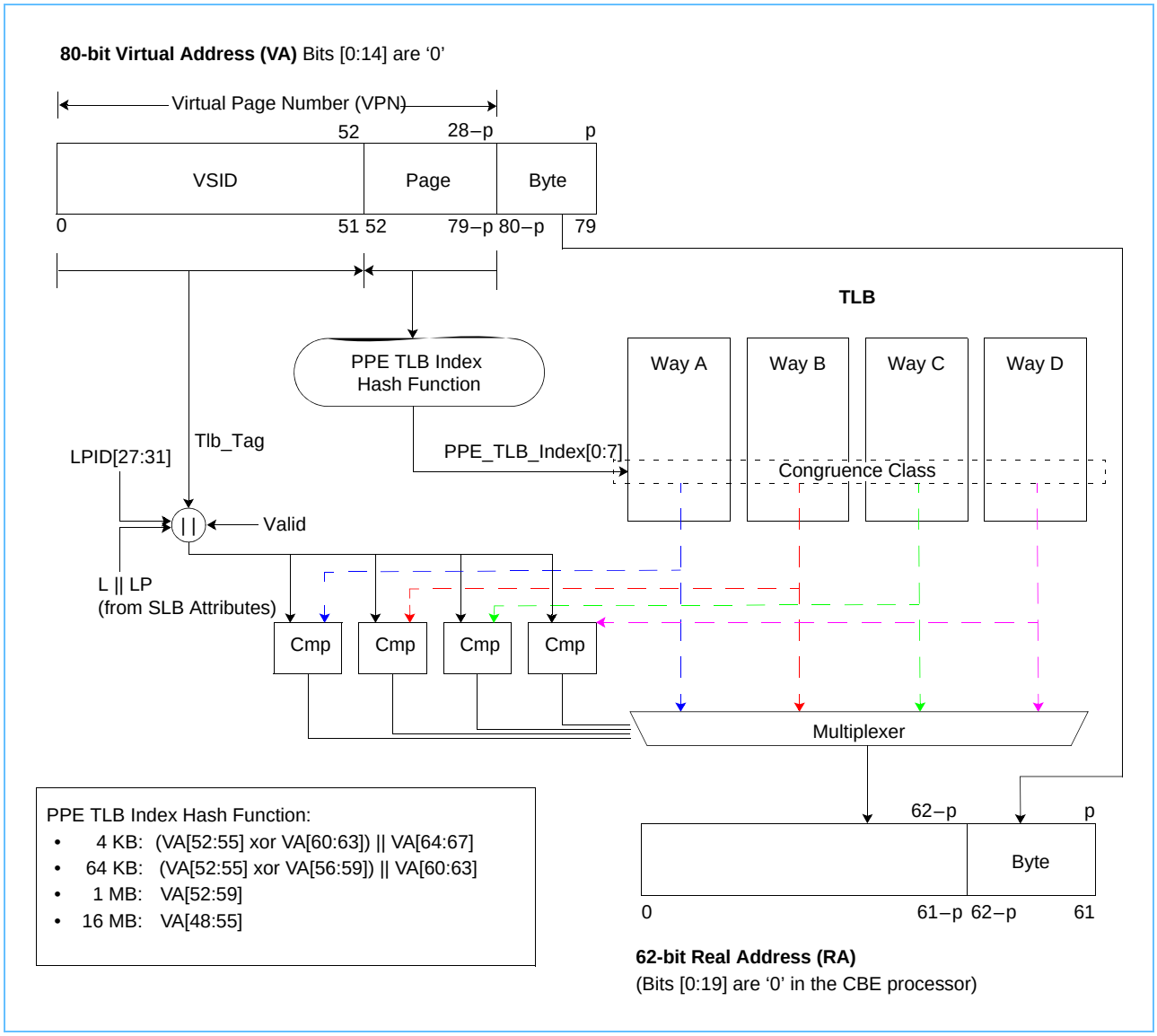
Translation Lookaside Buffer (TLB) entries and reloads can be managed either by hardware or by hypervisor software. The hardware-managed TLB mode is enabled when the TLB Load (`TL`) bit of the Logical Partition Control Register is cleared to ‘0’ (`LPCR[TL]` = ‘0’), a bit that is not defined as such in the *PowerPC Architecture*. The software-managed TLB mode is enabled when `LPCR[TL]` = ‘1’. In the software mode, the hypervisor uses MMIO-register accesses to manage the TLB as described in *Section 4.2.7.4 Software Management of the TLB* on page 95.

7. Misaligned, caching-inhibited stores are speculative because the instruction is not yet committed; an alignment interrupt has been taken, and the table lookup has not yet been performed at the time of the store.

4.2.7.2 *TLB Lookup*

The result of an SLB lookup is a VA with segment attributes. This information is passed to the TLB to be converted to an RA with page attributes. *Figure 4-3* shows the VA-to-RA conversion performed by a TLB entry in the hardware-managed TLB mode. The VA is split to form the Page field, which is input to an Index Hash Function, and the VSID field, which is used as a compare tag (Tlb_Tag). The result of the Index Hash Function produces the index (PPE_TLB_Index) that chooses among the 256 congruence classes (sets, or rows) and the four ways (columns) of the TLB.

Figure 4-3. Mapping a Virtual Address to a Real Address



The compare tag (Tlb_Tag) also contains the LPID, Valid (V, set to ‘1’), Large Page (L), and Large Page Selector (LP) fields of the PTE. Both L and LP come from the SLB segment attributes. The compare tag is compared against the data stored in each way of the indexed congruence class. The width of the compare tag changes for each page size, as shown in *Table 4-5*.

Table 4-5. TLB Compare-Tag Width

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

If a hit results from the compare, the entry for the TLB way (see *Figure 4-3* on page 93) is read out through the multiplexer. This data contains a previously cached PTE corresponding to the VA presented to the TLB, which holds the RPN and page attributes.

If this is an instruction-address translation, the page attributes are checked to ensure that the page is not marked Guarded (G = ‘1’) or No-execute (N = ‘1’), plus all other architected checks to ensure that access is allowed. If any condition is violated, an Instruction Storage interrupt is generated. Otherwise, if this is a data transaction or if the conditions are not violated, the RPN and page attributes are returned to the PPE’s ERAT.

4.2.7.3 *TLB Replacement Policy*

If a TLB miss results from the above compare operation, and hardware replacement is enabled (*Section 4.2.7.1* on page 92), a pseudo-LRU algorithm is used to replace the TLB entry. The replacement policy is a 3-bit pseudo-LRU policy in binary-tree format. It has 256 entries, each representing a single congruence class in the TLB. The pseudo-LRU algorithm used for PPE TLB replacement is identical to that used for SPE TLB replacement, as described in *Section 4.3.5.2 Hardware TLB Replacement* on page 109.

The LRU uses the following basic rules:

- The LRU is updated every time an entry in the TLB is hit by a translation. The entry that was referenced becomes the most-recently used, and the LRU bits are updated accordingly. However, the LRU is not updated when software reads the TLB for debug purposes using **mfspr** `PPE_TLB_VPN` or **mfspr** `PPE_TLB_RPN`.
- On a TLB miss, the update algorithm determines which entry to replace, using the following criteria:
  - *Invalid Entry*—Any invalid entry is replaced. If more than one entry is invalid, the left-most (lowest-numbered) way is chosen as invalid.
  - *Valid Entry*—The entry that the Replacement Management Table (RMT) determines to be both least-recently used and eligible for replacement is replaced.

For a description of the Replacement Management Table facility, see *Section 4.2.7.7 TLB Replacement Management Table* on page 98.

4.2.7.4 *Software Management of the TLB*

TLB reloading can operate in either a hardware-managed or software-managed TLB mode (see *Section 4.2.7.1* on page 92). In the hardware-managed mode, described above, software need not keep the TLB consistent with the hardware-accessed page table in main storage. In the software-managed mode, TLB management instructions and MMIO registers directly specify VA-to-RA translations, without the use of a hardware-accessed page table in main storage.

Software management of the TLB is restricted to the hypervisor, because memory is a system resource. Any attempt to execute TLB-management instructions when `MSR[PR, HV]` are not equal to '01' causes an Illegal Instruction interrupt.

The PPE implements the following MMIO registers, defined in the *Cell Broadband Engine Architecture*, for software management of the TLB:

- PPE Translation Lookaside Buffer Index Hint Register (`PPE_TLB_Index_Hint`).
- PPE Translation Lookaside Buffer Index Register (`PPE_TLB_Index`).
- PPE Translation Lookaside Buffer Virtual-Page Number Register (`PPE_TLB_VPN`).
- PPE Translation Lookaside Buffer Real-Page Number Register (`PPE_TLB_RPN`).

The PPE does not implement the following registers:

- TLB Invalidate Entry Register (`TLB_Invalidate_Entry`).
- TLB Invalidate All Register (`TLB_Invalidate_All`).

Software management of the TLB allows the system-software designer to forego the requirement of a hardware-accessible page table and use any page-table format. The format and size of a software-managed page table are not restricted by hardware. Software interprets the page table in order to load the TLB in response to a Data Storage interrupt caused by a TLB miss. Software TLB management can also be used in combination with hardware management to preload translations into the TLB.

One difference between a hardware and software TLB reload is the point at which the Data Storage interrupt (DSI) or SPE interrupt is presented to the PPE. For a hardware TLB reload, the interrupt is generated only after the page table is searched by hardware and a translation is not found. For a software TLB reload, the interrupt is generated when a translation is not found in the TLB.

In the software-managed TLB mode, all TLB misses result in a page fault leading to a Storage interrupt. Software can then write a new entry to the TLB using the code sequences and MMIO registers shown in *Table 4-6*. Software can also use this facility to preload the TLB with translation entries that are anticipated in future code execution. This is useful for avoiding TLB misses altogether.

Software should use the **tlbie** instruction to invalidate entries in the TLB, as described in *Table 4-6* and *Section 4.2.7.6 Software TLB Invalidation* on page 97. In *Table 4-6*, the command names in parentheses are recommended compiler mnemonics.

Table 4-6. TLB Software Management

TLB Operation[1]	Code Sequence

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

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

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

Invalidate an entry	`tlbie(l)`[2]
Invalidate all entries	See *Section 4.2.7.6 Software TLB Invalidation* on page 97.

1. Recommended compiler mnemonics are shown in parentheses.
2. See *Section 4.2.7.6 Software TLB Invalidation* on page 97.

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

The TLB tags each entry with the PPE Logical Partition ID (`LPID`) value at the time the entry is written. The hypervisor should set the `LPID` register to the value it wishes to have the TLB entry tagged with at the time the entry is written. See *Section 4.2.7.8 Logical-Partition ID Tagging of TLB Entries* on page 98 for details. Software must set bit 56 of the `PPE_TLB_VPN` register (the low-order bit of the `AVPN` field) to ‘0’ when writing a 16-MB TLB entry.

Reading the `PPE_TLB_VPN` or `PPE_TLB_RPN` register returns the PTE data associated with the TLB entry pointed to by the last-written value of `PPE_TLB_Index`. Reading the `PPE_TLB_Index_Hint` register returns the location that hardware would have chosen to replace when the last TLB miss occurred as a result of a translation for the thread performing the read operation (`PPE_TLB_Index_Hint` is duplicated per thread). If no miss has occurred, the register returns its POR value (typically all ‘0’s).

The ability to read the TLB is provided for hypervisor debugging purposes only. The `LPID` and `LVPN` fields of the TLB entry are not accessible to the hypervisor, because they are not defined in the PTE that the `PPE_TLB_VPN` and `PPE_TLB_RPN` registers reflect. This makes it impossible to save and restore the TLB directly.

Bit 56 of the `PPE_TLB_VPN` register (the low-order bit of the `AVPN` field) is undefined when the TLB entry is a 1-MB page.

4.2.7.5 *Rules for Software TLB Management*

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

- Both threads should not attempt to write the TLB at the same time. Doing so can lead to boundedly undefined results, because there is a nonatomic sequence of register updates required for TLB writes (even though the write to the TLB itself is atomic). For example, the following situation should be avoided: thread 0 writes the `PPE_TLB_Index` and `PPE_TLB_RPN` registers; then thread 1 writes over the `PPE_TLB_Index` register; then thread 0 writes the

PPE_TLB_VPN register. The result is that the wrong TLB entry is written because thread 1 caused the PPE_TLB_Index register to point to the wrong location in the TLB. An easy way to solve this problem is for the hypervisor to reserve a lock when writing the TLB.

- Both threads should not attempt to read the TLB at the same time, because reading the TLB using the **mtspr** and **mfspr** instructions requires first writing the PPE_TLB_Index register, so Rule • applies. In other words, the same hypervisor lock that is used to write the TLB should also be acquired to read the TLB using the **mtspr** and **mfspr** instructions. This prevents one thread from corrupting the PPE_TLB_Index pointer that the other thread is using to read the TLB. Failure to acquire a lock can lead to boundedly undefined results.

  Also, because reading the TLB using the **mtspr** and **mfspr** instructions is a nonatomic sequence, hardware updates of the TLB by the other thread should be prevented. For example, the following situation should be avoided: thread 0 reads the PPE_TLB_RPN register; then thread 1 performs a hardware table-lookup update of the same TLB entry; then thread 0 reads the PPE_TLB_VPN register. The PPE_TLB_RPN and PPE_TLB_VPN results will not correspond in this case. Because reading of the TLB using the **mtspr** and **mfspr** instructions is intended for debug use only, the hypervisor should stop the other thread from accessing the TLB while performing this operation.
- Locking is not required to translate in the TLB. Because writes to the TLB itself are atomic, the thread translating in the TLB sees either the previous or the new TLB entry, but never an intermediate result. This is analogous to hardware updates of the TLB.
- The TLB is a noncoherent cache for the page table. If the TLB is completely software-managed, then it is software's responsibility to ensure that all TLB contents are maintained properly and that coherency with the page table is never violated.

  Invalidating a TLB entry requires use of the **tlbie** or **tlbiel** instruction (see *Section 4.2.7.6* on page 97). This is required so that the ERATs maintain coherency with the TLB. This means, for example, that simply writing the Valid bit to ‘0’ for a TLB entry is not sufficient for removing a translation. Only the **tlbie(l)** instruction properly snoop-invalidates the ERATs.
- Software must maintain Reference (R) and Change (C) bits in the page table. See *Section 4.2.6.8* on page 92 for R and C bit maintenance by the hardware table-lookup. If software writes a TLB entry with C = ‘0’, a subsequent store operation in the software-managed TLB mode takes a page fault. The R bit is ignored by the TLB, so writing R = ‘0’ is ignored and treated like R = ‘1’.

Failure of the hypervisor to properly synchronize TLB read and write operations can lead to undefined results throughout the entire system, requiring a system reset.

4.2.7.6 *Software TLB Invalidation*

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

The PPE supports bits [22:(63 – *p*)], of the RB source register for **tlbie** and **tlbiel**, which results in a selective invalidation in the TLB based on VPN[38:(79 – *p*)] and the page size, *p*[8]. In other words, any entry in the TLB that matches the VPN[38:(79 – *p*)] and the page size is invalidated.

8. The value “p” is a power-of-2 variable in the *PowerPC Architecture* representing the size of a page.

The PPE adds new fields to the RB register of the **tlbiel** instruction that are not currently defined in the *PowerPC Architecture*. These include the Large Page Selector (LP) and Invalidation Selector (IS) fields. The IS field is provided in RB[52:53] of the **tlbiel** instruction. *Table 4-7* gives details of the IS field. Bit 1 of the IS field is ignored.

Table 4-7. Supported Invalidation Selector (IS) Values in the tlbiel Instruction

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

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

4.2.7.7 *TLB Replacement Management Table*

The PPE provides a method of controlling the TLB replacement policy based on a Replacement-class ID (RclassID). Software can use the Replacement Management Table (RMT) stored in the PPE Translation Lookaside Buffer RMT Register (`PPE_TLB_RMT`) to lock translation entries into the TLB by reserving a particular way of the TLB to a specific program EA range. For details, see *Section 6.3 Replacement Management Tables* on page 154.

4.2.7.8 *Logical-Partition ID Tagging of TLB Entries*

The TLB supports entries for multiple logical partitions simultaneously. Each entry in the TLB is tagged with a 5-bit Logical Partition ID (`LPID`), which is specified in the Logical Partition Identity Register (`LPIDR`). This tag is assigned when the TLB entry is updated by either hardware or software.

The `LPID` tag on each TLB entry is used by hardware for the following purposes:

- To identify a TLB entry as belonging to a given partition. The TLB does not need to be saved, invalidated, and restored on a logical-partition switch, thus supporting high-performance partition-switching.
- To determine if a TLB entry is a match for a translation request. To be considered a match, the current value of the processor element’s `LPID` register and the tag in the TLB must be equivalent. If not, a TLB miss occurs.
- To determine if a **tlbie** should invalidate a given TLB entry. Each processor appends the current `LPID` information to every **tlbie** sent out to the bus. When a **tlbie** is received by the TLB, the `LPID` information can be used to selectively invalidate only the TLB entries that correspond to the same partition as the **tlbie**.

It is possible for one partition to invalidate or discard the TLB entries of another partition by issuing a **tlbie** instruction with IS set to ‘11’. See *Section 4.2.7.6* on page 97 for details.

Software can read TLB entries, but software cannot read the value of a TLB entry's LPID tag. Theoretically, this allows one partition to read the TLB entries of another partition, but it does not allow the partition to know which partition owns the TLB entry. However, reading from the TLB is restricted to the hypervisor, so the hypervisor should not provide a service that allows operating systems or applications to read the TLB-entry values. Reading TLB entries is not atomic, is low performance, and can cause a security risk. It is intended for hypervisor software debugging only.

Processor elements will ignore **tlbie** instructions that do not have an `LPID` value matching the processor element's current `LPIDR` value. Because of this, when moving a partition from one processor element to another, software must invalidate all TLB entries on the current processor element before changing the `LPIDR`. The `LPID` tagging only benefits partition-switching on a single processor element, not switching a partition between multiple processor elements. The Invalidation Selector (IS) field of the **tlbiel** instruction can be used to override this behavior. If IS = '11', then the TLB performs a congruence-class invalidation of the TLB without regard to the `LPID` value supplied.

The `LPID` is ignored by the TLB replacement logic when performing a hardware table-lookup.

4.2.8 Real Addressing Mode

If the PPE operating system does not enable virtual addressing, as described in *Section 4.2.3 Enabling Address Translation* on page 83, the system operates in *real addressing mode* (also called *real mode*). In this mode, EAs of application programs running on the PPE are used as RAs (possibly offset by an `RMOR` or `HRMOR` base address) and there is no page table or paging. In contrast to this PPE behavior, real-mode addresses generated by an SPE's MFC are never offset by `RMOR` or `HRMOR` base-address registers and they do not use the `MSR[HV]` and `LPCR[LPES]` fields and the `RMLR` register, as described in *Section 4.3.6* on page 116.

All real-mode accesses are treated as if they belong to a 4-KB page. If a PPE system has a smaller physical-address range than a program's EA range, the extra high-order bits of the EA are ignored (treated as if they were zeros) in the generation of the RA.

4.2.8.1 *Types of Real Addressing Modes*

The contents of the following registers affect the real addressing mode, including real-mode accesses for page-table lookups:

- Real Mode Offset Register (`RMOR`).
- Hypervisor Real Mode Offset Register (`HRMOR`).
- The `RMLS`, `RMI`, and `LPES` (bit 1) fields of the Logical Partition Control Register (`LPCR`).
- The Real-Mode Storage Control (`RMSC`) field of the Hardware Implementation Register 6 (`HID6`).

Table 4-8 summarizes how the RA is determined from the EA in real mode. An address-translation fault causes an Instruction Storage or Data Storage interrupt. For further details, see the Real Addressing Mode section of the *PowerPC Operating Environment Architecture, Book III*.

Table 4-8. Summary of Real Addressing Modes

Mode Name	Mode Bits	Real-Address Calculation
Hypervisor Offset Mode	MSR[HV] = '1' EA(0) = '0'	RA = (EA[22:43] | HRMOR[22:43]) || EA[44:63]
Hypervisor Real Mode	MSR[HV] = '1' EA(0) = '1'	RA = EA[22:63]
Real Offset Mode	MSR[HV] = '0' LPCR[LPES] bit 1 = '1'	RA = (EA[22:43] | RMOR[22:43]) || EA[44:63]
Mode Fault	MSR[HV] = '0' LPCR[LPES] bit 1 = '0'	LPAR Error (Interrupt)

For details on how the WIMG bits are set in real mode, see *Section 4.2.6.7 WIMG-Bit Storage Control Attributes* on page 91. Other attribute bits are set in real mode as shown in *Table 4-9*.

Table 4-9. Summary of Real-Mode Attributes

Attribute Bit	Value	Attribute Description
L	0	Large Page
LP	0	Large Page Selector
Ks	0	Supervisor State Storage Key
Kp	0	Problem State Protection Key
CL	0	Class
AC	0	Address Compare
PP[0:1]	10	Page Protection
C	1	Change
N	0	No Execute

4.2.8.2 ***Real-Mode Storage Control Facility***

The PPE supports a Real-Mode Storage Control (`RMSC`) facility. This facility allows for altering the storage-control attributes in real mode based on a boundary in the real-address space. The value in the 4-bit RMSC field stored in `HID6[26:29]` determines the boundary as shown in *Table 4-10*.

Table 4-10. Real-Mode Storage Control Values (Page 1 of 2)

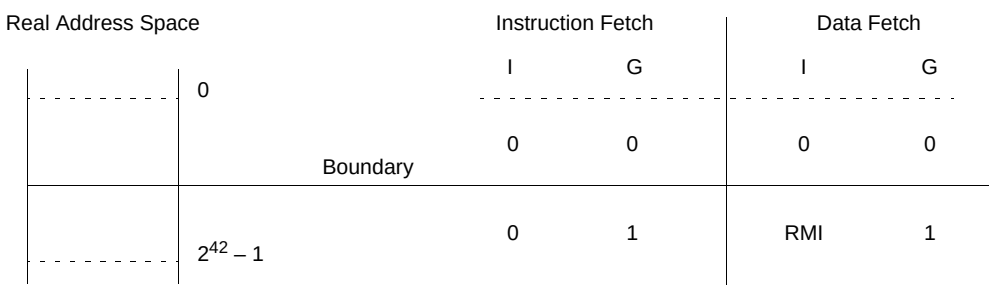
Value in HID6[RMSC] Register Field	Real-Address Boundary
0000	0
0001	256 MB
0010	512 MB
0011	1 GB
0100	2 GB
0101	4 GB
0110	8 GB
0111	16 GB

Table 4-10. Real-Mode Storage Control Values (Page 2 of 2)

Value in HID6[RMSC] Register Field	Real-Address Boundary
1000	32 GB
1001	64 GB
1010	128 GB
1011	256 GB
1100	512 GB
1101	1 TB
1110	2 TB
1111	4 TB

Figure 4-4 demonstrates how the I (caching-inhibited) and G (guarded) real-mode storage-attribute bits are set according to the location of the Real Address of the storage access relative to the boundary. “RMI” is the Real-Mode Caching Inhibited bit in `LPCR[62]`.

Figure 4-4. Real-Mode I and G Bit Settings



4.2.8.3 ***Real-Mode Maintenance***

Real-mode translations are kept in the ERAT, so software must issue the following sequence when changing any of the `RMOR`, `HRMOR`, `LPCR`, or `HID6` registers:

- **mtspr** for `RMOR`, or `HRMOR`, or `LPCR[RMLS, RMI, LPES bit 1]`, or `HID6[RMSC]`.
- **isync**.
- **tlbiel** L = ‘1’, IS = ‘00’, RB = ‘0’.
- **sync** L = ‘1’.

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

Changing any of the above registers changes the context of all real-mode translations, so care must be taken to ensure that any previous real-mode translations are completed from a storage perspective and are invalidated from the ERAT. The above sequence should also be followed when multiple registers are to be changed at once—for example, when switching between logical partitions, the `RMOR`, `RMLS`, and `LPIDR` register need to by changed at once. When changing any of these register values, software must not cause an implicit branch, as described in the Implicit Branch section of *PowerPC Operating Environment Architecture, Book III*.

4.2.9 Effective Addresses in 32-Bit Mode

The PPE supports both the 64- and 32-bit memory management models defined by the *PowerPC Operating Environment Architecture, Book III*. The *PowerPC Architecture* also defines a 32-bit mode of operation for 64-bit implementations, which the PPE supports.

In this 32-bit mode (`MSR[SF]` = ‘0’), the 64-bit EA is first calculated as usual, and then the high-order 32 bits of the EA are treated as ‘0’ for the purposes of addressing memory. This occurs for both instruction and data accesses, and it occurs independently from the setting of the `MSR[IR]` and `MSR[DR]` bits that enable instruction and data address translation, respectively. The truncation of the EA is the only way in which memory accesses are affected by the 32-bit mode of operation.

4.3 SPE Memory Management

All information in main storage is addressable by EAs generated by programs running on the PPE, SPEs, and I/O devices. An SPE program accesses main storage by issuing a DMA command, with the appropriate EA and LS addresses. The EA part of the DMA-transfer address-pair references main storage. Each SPE’s MFC has two command queues with associated control and status registers. One queue—the MFC SPU Command Queue—can only be used by its associated SPE. The other queue—the MFC Proxy Command Queue—can be mapped to the EA address space so that the PPE and other SPEs and devices can initiate DMA operations involving the LS of the associated SPE.

When virtual addressing is enabled by privileged software on the PPE, the MFC of an SPE uses its Synergistic Memory Management (SMM) unit to translate EAs from an SPE program into RAs for access to main storage. The SMM’s functions are similar to those of the PPE’s MMU described in *Section 4.2 PPE Memory Management* on page 80, except that the PPE provides an SMM with segment-buffer and page-table information needed for address translation. This section describes the differences between the SPE’s SMM and the PPE’s MMU.

4.3.1 Synergistic Memory Management Unit

Address translation is managed in the SMM by the following hardware units:

- *Segment Lookaside Buffer (SLB)*—A unified (instruction and data), 8-entry array that provides EA-to-VA translations. The SLB is mapped to main storage as MMIO registers. The SPE supports up to 2^{37} segments of 256 MB each.
- *Translation Lookaside Buffer (TLB)*—The TLB is a unified (instruction and data), 256-entry, 4-way set-associative cache that stores recently accessed PTEs supplied by the PPE operating system.

Table 4-11 summarizes the features of the SMM unit.

Table 4-11. SPE Synergistic Memory Management Unit Features (Page 1 of 2)

Parameter		SPE Implemented Value or Feature
Addresses	Effective Address (EA) Size	Same as PPE.
	Virtual Address (VA) Size	
	Real Address (RA) Size	

Table 4-11. SPE Synergistic Memory Management Unit Features (Page 2 of 2)

Parameter		SPE Implemented Value or Feature
SLB	SLB Entries	8
	Segment Size	256 MB
	Number of Segments	2^{37}
	Segment Protection	None. The Class (c) and No-execute (N) bits are stored in the SLB but are not used within the SMM.
	SLB Maintenance	Special instructions for maintaining SLB.
Page Table	Number of Page Tables	The SMM uses page-table entries supplied by the PPE.
	Table Structure	
	Page Protection	
	Page History	
	Page Sizes	4 KB, plus two of the following Large Page Sizes: • 64 KB (p = 16). • 1 MB (p = 20). • 16 MB (p = 24).
	Number of Pages	Determined by MFC_SDR[HTABSIZE] register field. Page sizes specified in this field must agree with the PPE SDR1[HTABSIZE] register field for the same page table.
TLB	TLB Entries	256
	TLB Maintenance	• Hardware- or software-managed TLB update. • Maintained by accesses to MMIO registers. • Pseudo-LRU replacement policy. • Resource management table (RMT) for TLB replacement.

4.3.2 Enabling Address Translation

Virtual-address translation is enabled in an SPE by setting the Relocate (R) bit in the MFC State Register 1 (`MFC_SR1[R]`) to ‘1’. When this is done, the SLB, TLB, and page table facilities in the MFC are used to translate EAs in MFC commands to VAs, and then to a RAs.

If `MFC_SR1[R]` = ‘0’, the main-storage side of MFC accesses operates in real mode. In this mode, EAs are used as RAs, but unlike PowerPC real addresses, the MFC real addresses are never offset by an RMOR or HRMOR base address. In addition, a page table is not used in real mode, and there is no demand-paging, as described in *Section 4.2.8 Real Addressing Mode* on page 99.

The following sections describe details of SMM address translation. Additional details are given in the Privileged-Mode Environment section of the *Cell Broadband Engine Architecture*, *Cell Broadband Engine Registers*, and the *PowerPC Operating Environment Architecture, Book III*.

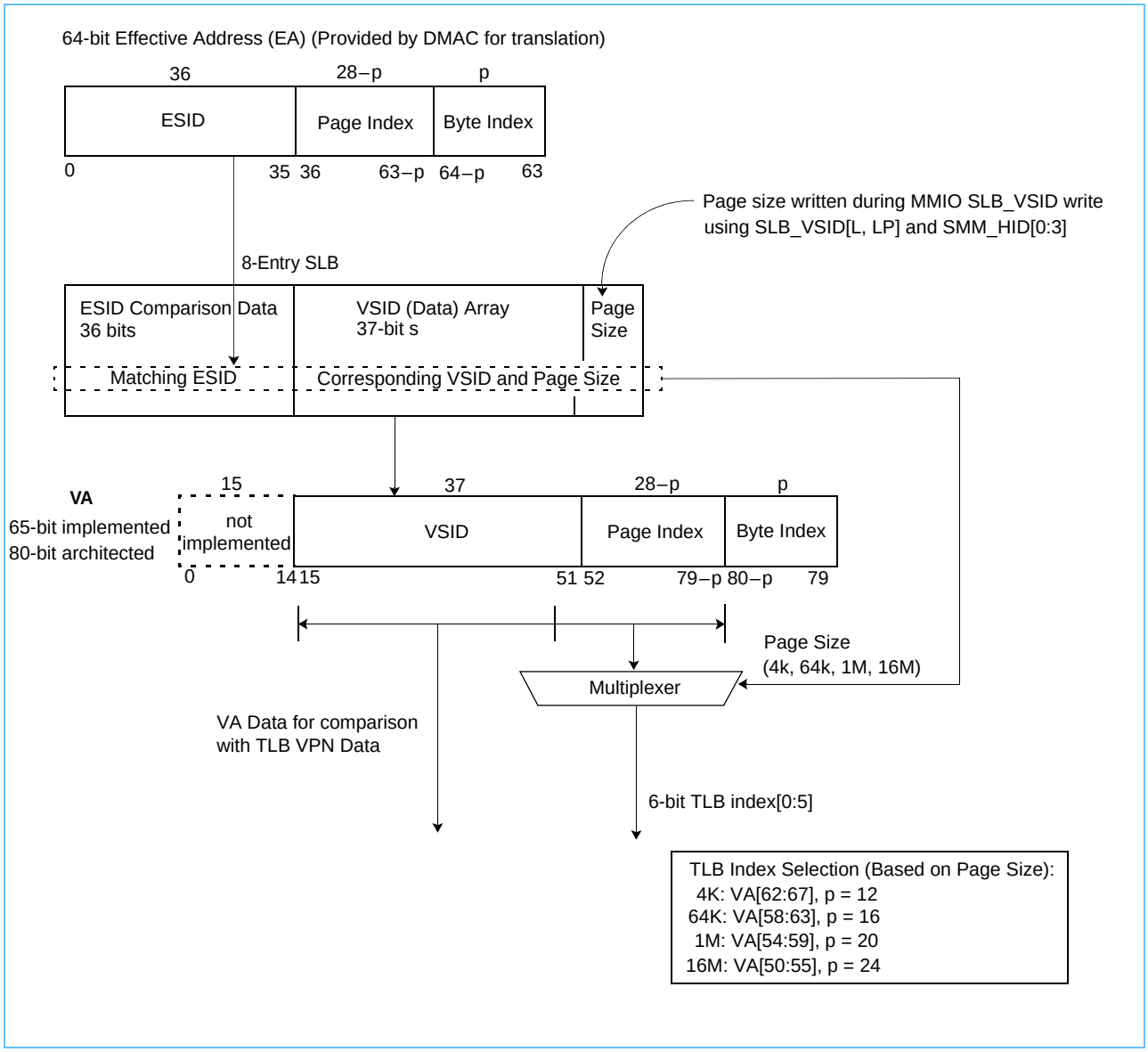
4.3.3 Segmentation

The SMM implements an 8-entry SLB. The SLB translates Effective Segment IDs (ESIDs) to Virtual Segment IDs (VSIDs)—which also translates EAs to VAs. The SMM has a set of MMIO registers for management of its SLB. These registers provide the same functionality for the SMM as do PowerPC instructions for the PPE's MMU, described in *Section 4.2.5 Segmentation* on page 85.

The SLB's Virtual Segment ID (`VSID`) field implements 37 bits of the architecture-defined 52-bit `VSID`. The Class (`C`) and No-execute (`N`) bits are stored in the SLB array but are not used within the SMM. All portions of the SLB, except the ESID and Valid bit, are protected by parity. Parity generation and checking can be disabled in the `SMM_HID` register.

Unlike the PPE's MMU, the SMM uses only a direct-mapped TLB—thus, all pages of the same size use the same virtual-address bits to obtain the TLB index for TLB lookups. Due to timing constraints, TLB-index selection occurs within the SLB array. Page-size encoding is saved in a separate 4-bit field of the SLB. The SMM uses six SLBs of the Virtual Page Number (VPN) selected by the encoded page size to select a TLB entry. The page size is set and stored in the array on a write to the SLB Virtual Segment ID Register (`SLB_VSID`), and it is determined by the `L` and `LP` bits in the `SLB_VSID` data and the page-size encodings in the SMM Hardware Implementation Dependent Register (`SMM_HID`). See *Table 4-13* on page 112 for information on setting the page size. *Figure 4-5* illustrates the EA-to-VA mapping and TLB index selection.

Figure 4-5. SPE Effective- to Virtual-Address Mapping and TLB-Index Selection



4.3.3.1 *SLB Registers*

The SMM has a set of MMIO registers for SLB management. These registers provide the same functionality for the SMM that is provided by the *PowerPC Architecture* instructions (**slbie**, **slbia**, **slbmte**, **slbmfev**, and **slbmfee**) for managing the PPE's SLB. The SLB-management registers include:

- SLB Index Register (`SLB_Index`).
- SLB Effective Segment ID Register (`SLB_ESID`).
- SLB Virtual Segment ID Register (`SLB_VSID`).
- SLB Invalidate Entry Register (`SLB_Invalidate_Entry`).

- SLB Invalidate All Register (`SLB_Invalidate_All`).

The section below provide detail. See the *Cell Broadband Engine Registers* document for the implemented bits of these *Cell Broadband Engine Architecture* registers.

4.3.3.2 *SLB-Entry Replacement*

The SLB requires a write sequence, similar to the one required for the TLB, to properly load the SLB with data. The SLB MMIO registers must be accessed by privileged PPE software with storage attributes of caching-inhibited and guarded.

First, the index must be written to specify the entry for loading. The `VSID` and `ESID` fields must be written independently, unlike the TLB writes, but the `SLB_VSID` write should follow the index. The `SLB_ESID` data is written last because it contains the Valid bit, and the entry should not be valid until all data is loaded. The `SLB_Index` register is write-only. The `SLB_ESID` and `SLB_VSID` registers are both read-write; however, MMIO reads retrieve data from the array instead of the architected register.

The sequence to replace an SLB entry is:

1. For each entry to be replaced:
   a. Set the index of the SLB entry to be replaced.
   b. Use the `SLB_Invalidate_Entry` register to invalidate the SLB entry.
2. Set the new contents for the `VSID` portion of the SLB entry.
3. Set the new contents for the `ESID` portion of the SLB entry along with the valid bit.

The contents of an SLB entry are accessed by using the `SLB_ESID` and `SLB_VSID` registers. The `SLB_Index` register points to the SLB entry to be accessed by the `SLB_ESID` and `SLB_VSID` registers.

4.3.3.3 *SLB Mappings and Interrupts*

Software must maintain a one-to-one mapping between ESID and VSID entries, because multiple hit-detection is not supported by the SLB. The bits in the VSID are ANDed together if there are multiple matching ESIDs, and the results are unpredictable. If an SLB lookup fails due to an EA mismatch or cleared Valid bit, the SMM asserts a Segment interrupt. All other DMA translation-request interrupts on a single translation are masked by a Segment interrupt.

The SMM has a non-blocking, hit-under-miss pipeline architecture. The SMM can handle further DMA-translation hits under a pre-existing interrupt. Any subsequent interrupts are masked, and that translation is returned as a miss to the DMAC until the original interrupt is cleared. After the original interrupt is cleared, the subsequent interrupt can be seen.

4.3.3.4 *SLB Invalidation*

Software must use the `SLB_Invalidate_Entry` register to maintain the SLB and update entries. However, SLB invalidations occur by writing an EA in the `SLB_Invalidate_Entry` register that matches the `ESID` data in the entry to be replaced. Entries are invalidated by clearing the Valid bit in the `ESID` field. The remainder of the `ESID` and `VSID` data is untouched for register-readout debugging purposes.

SLB_Invalidate_Entry EA data is written into the SLB_ESID register because they share EA format data. The SLB_Invalidate_Entry register is write-only, and the SLB_Invalidate_Entry data cannot be retrieved because SLB_ESID reads return data from the SLB array and not the architected register. Reads from this register return ‘0’ data.

No data is retained for writes to the SLB_Invalidate_All register. The write action prompts the SLB array to clear all eight Valid bits in the array and, therefore, invalidate all registers. This is a write-only register. Reads from this register return ‘0’ data.

4.3.3.5 *SLB After POR*

All SLB entries and TLB entries are invalidated after Power-On Reset (POR).

4.3.4 Paging

4.3.4.1 *Page Table*

An SPE has no direct access to system-control facilities, such as the page table or page table entries (PTEs) for a logical partition. This restriction is enforced by PPE privileged software. The SPE’s Atomic (ATO) Unit provides PPE-generated PTE data to the SMM for hardware table-lookups, as needed.

4.3.4.2 *Large Pages*

The page size can be 4 KB or any two of the three Large Page Sizes: 64 KB, 1 MB, or 16 MB. The selection between the two large page sizes is controlled by:

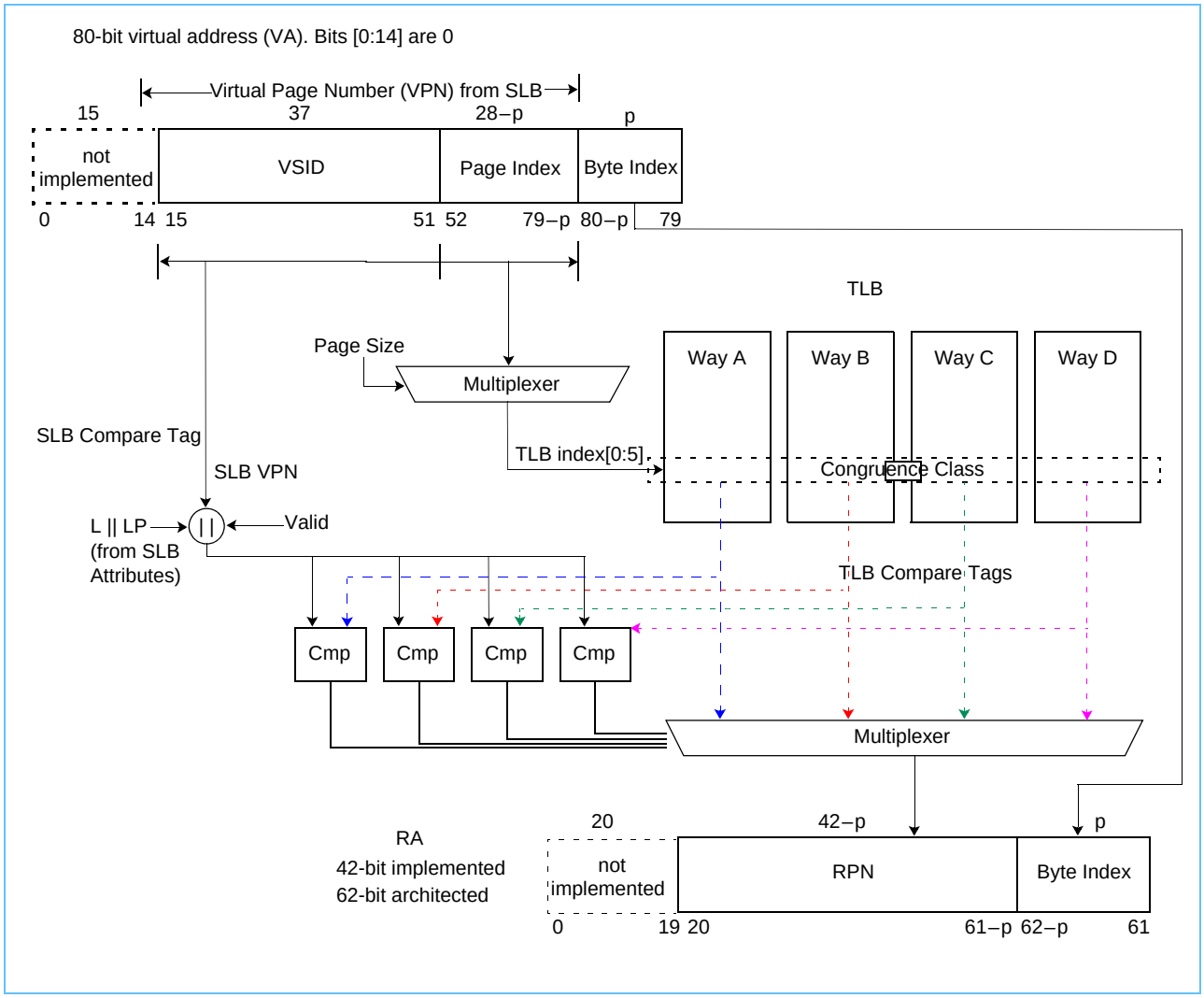
- Page Size Decode (Pg_Sz_Dcd) field of the SMM Hardware Implementation Dependent Register (SMM_HID). See *SMM_HID Register* on page 112.
- Large Page (L) bit of the MFC TLB Virtual-Page Number Register (MFC_TLB_VPN[L]).
- Large Page (LP) field of the MFC TLB Real Page Number Register (MFC_TLB_RPN[LP]).
- Large Page (L) bit and Large Page-Size (LP) bit of the SLB Effective Segment ID Register (SLB_ESID[L, LP]).

See *Table 4-13* on page 112 for large-page-size decoding.

4.3.5 Translation Lookaside Buffer

The TLB is a 256-entry, 4-way, set-associative cache that holds most-recently used PTEs that are provided by the PPE. There are 64 congruence classes (sets), each requiring six bits for the TLB index. When a translation is not found in the TLB, the SMM permits either a hardware or software TLB reload, as specified in the MFC State Register 1 (MFC_SR1). The TLB array is parity protected. Parity generation and checking can be disabled in the SMM_HID register. *Figure 4-6* on page 108 illustrates the VA-to-RA mapping.

Figure 4-6. SMM Virtual- to Real-Address Mapping



The 6-bit TLB index selects four PTE entries (ways) in a congruence class. The four `VPN` fields (the TLB Compare Tags) of the selected entries are compared to the `VPN` field obtained from SLB lookup (the SLB Compare Tag). *Table 4-12* on page 109 shows the SLB and TLB Compare Tag definitions for each page size. The TLB `VPN` data is a concatenation of `MFC_TLB_VPN` and `MFC_TLB_Index` register data during MMIO TLB writes. The TLB `RPN` data is a concatenation of the `MFC_TLB_RPN[ARPN]` and `MFC_TLB_RPN[LP]` register fields.

For large pages in which the L bit is set to ‘1’, the RPN field concatenates the `ARPN` range with the upper seven bits of the `LP` field and a value of ‘0’. The lower bit of the `LP` field is used to select between the two large pages indicated by the `SMM_HID` register. The lower bit of the `LP` field is also used as part of the TLB compare tag, as shown in *Table 4-12*. If the L bit is not set to ‘1’, the `RPN` field is formed by concatenating the `ARPN` field with the whole `LP` field. See *MFC_TLB_Index, MFC_TLB_RPN, and MFC_TLB_VPN Registers* on page 113 for more information.

Table 4-12. SLB and TLB Compare Tags

Page Size	SLB Compare Tag	TLB Compare Tag
4 K	VA[15:61] || Valid || L || LP	VPN[15:61] || Valid || L || LP
64 K	VA[15:57] || Valid || L || LP	VPN[15:57] || Valid || L || LP
1 M	VA[15:53] || Valid || L || LP	VPN[15:53] || Valid || L || LP
16 M	VA[15:49] || Valid || L || LP	VPN[15:49] || Valid || L || LP

4.3.5.1 *Enabling Hardware or Software TLB Replacement*

Hardware or software can be used to replace the TLB entry. The mode is selected by the `TL` bit of MFC State Register 1 (`MFC_SR1`).

4.3.5.2 *Hardware TLB Replacement*

When there is a TLB-lookup miss, the SMM determines which TLB entry to replace by using a pseudo-LRU algorithm and a Replacement Management Table (RMT), as described in *TLB Replacement Policy* on page 110. That data is saved in the `MFC_TLB_Index_Hint` register.

Replacement Method

If hardware replacement is selected, the SMM performs a hardware table-lookup to find the replacement data and write the TLB at the location indicated by the `MFC_TLB_Index_Hint` register. A PTE request from the SMM is basically a store request. See *Section 6.2 SPE Caches* on page 151 for background information about an SPE's TLB and atomic cache.

If the cache line containing the PTE is in the MFC's Atomic cache, which allocates a maximum of one line for PTEs, and the line is in the modified or exclusive cache state, the PTE is sent to the SMM. If the SMM requests an update for Reference (R) or Change (C) bits in the PTE, the Atomic Unit executes the store and updates those bits.

If the cache line containing the PTE is in the MFC's Atomic cache, and in a state other than modified or exclusive, a bus request is sent to invalidate copies of the data in other processor elements, and the PTE is sent to the SMM. If the SMM requests an update for R or C bits in the PTE, the Atomic Unit executes the store and updates those bits.

If the cache line containing the PTE is not in the Atomic Unit, a bus request is sent to invalidate copies of the data in other caches and to read the data. If the Atomic cache is full, the same castout scenario applies here as with the **putllc** or **putlluc** requests. That is, the PTE is sent to the SMM. If the SMM requests an update for R or C bits in the PTE, the Atomic Unit executes the store and updates those bits.

It is a software error if the PTE is used by the semaphore Atomic Unit operation as a locked line.

TLB Replacement Policy

The SMM uses a 3-bit, binary-tree pseudo-LRU algorithm to select a TLB entry for replacement after a TLB miss. There are 64 TLB congruence classes, each containing four (ways), thus 256 TLB entries as shown in *Figure 4-6* on page 108. A Replacement Management Table (RMT) can be used to lock entries and prevent their replacement, as described in *Section 6.3.3 SPE TLB Replacement Management Table* on page 158.

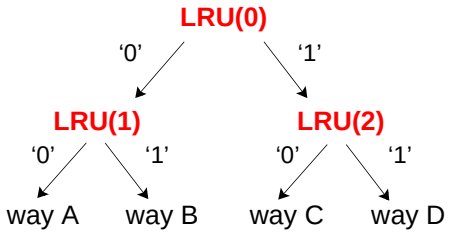
The following rules govern LRU use and TLB-entry replacement:

1. The LRU is updated every time an entry in the TLB is hit by a translation operation. The entry that was referenced becomes the most-recently used, and the LRU bits are updated accordingly. The LRU is not updated if that TLB entry is locked by the RMT.
2. On a TLB miss, the update algorithm determines which entry to replace, using the following priority:
   a. Replace the entry that hit in the TLB, but was a **put** command and the Change (C) bit was ‘0’.
   b. Replace the first entry (way) in the congruence class that was invalid (Valid bit was ‘0’).
   c. Replace the least-recently used entry, indicated by the pseudo-LRU algorithm, that is not locked by the RMT.

After a TLB entry is chosen for replacement using the rules above, the TLB replacement index, which specifies TLB entries as a congruence class and way, is saved in the `MFC_TLB_Index_Hint` register (see *MFC_TLB_Index_Hint Register* on page 113). The last four bits of the TLB_Index_Hint register are a one-hot vector to indicate the entry to be replaced. The bits represent the four TLB ways within a congruence class like the four bits on an RMT entry, described in *Section 6.3.3 SPE TLB Replacement Management Table* on page 158. The `MFC_TLB_Index_Hint` vector can only indicate the congruence-class way that is selected for replacement, whereas the RMT entry indicates all ways in the congruence class that are available for replacement.

Figure 4-7 shows the 3-bit pseudo-LRU binary tree. For each pseudo-LRU bit, a ‘0’ value moves to the left and a ‘1’ moves to the right. Only two bits are needed to determine one of the four congruence-class ways.

Figure 4-7. Pseudo-LRU Binary Tree for SPE TLB



Take, for example, an address translation with `RClassID` equal to x’2’ that misses in the TLB at index x’34’. The LRU would then be read at index x‘34’ to retrieve the three pseudo-LRU bits. If LRU(0) equals ‘1’, we follow the tree to the right, and if LRU(2) equals ‘0’, we follow the tree to the left and way C is chosen for replacement.

The RMT must be checked to see if way C has been locked by software. The `RClassID` value selects RMT entry 2. If the RMT entry equals ‘1110’, then way C is unlocked and can be replaced. If the RMT entry equals ‘1100’, however, LRU(0) and LRU(2) are overridden by the RMT, and LRU(1) chooses either way A or way B for replacement. If LRU(1) equals ‘1’, way B is chosen for replacement, the `MFC_TLB_Index_Hint` register is written with index x‘34’, and way B is written as follows: `MFC_TLB_Index_Hint[54:63]` equals ‘11_0100_0100’.

Interrupt Conditions

The SMM contains logic to detect and report interrupt conditions associated with TLB data, including atomic access to caching-inhibited pages (real mode and virtual mode), Page-Protection (PP) violations, Data Address Compare (DAC) errors, TLB page faults (hardware- and software-managed modes), and parity errors during DMA and MMIO-register accesses. Real-mode translation caching-inhibited errors are detected by the DMAC and reported to the SMM for Data Storage Interrupt (DSI) assertion. These errors are summarized in *Section 4.3.7 Exception Handling and Storage Protection* on page 117.

4.3.5.3 ***Software TLB Replacement***

As an alternative to the hardware-managed TLB mode (*Section 4.3.5.1 Enabling Hardware or Software TLB Replacement* on page 109), most SMM maintenance can be performed by PPE software accessing MMIO registers. If software replacement is enabled, privileged PPE software searches the PPE’s page table and provides a replacement using access to MMIO registers in the SMM. Software can use the `MFC_TLB_Index_Hint` data or choose another location within the TLB.

The SMM contains only Privilege 1 (hypervisor) and Privilege 2 (operating system) registers.

The TLB-management registers include:

- SMM Hardware Implementation Dependent Register (`SMM_HID`).
- MFC TLB Invalidate Entry Register (`MFC_TLB_Invalidate_Entry`).
- MFC TLB Replacement Management Table Data Register (`MFC_TLB_RMT_Data`).
- MFC Storage Description Register (`MFC_SDR`).
- MFC Address Compare Control Register (`MFC_ACCR`).
- MFC TLB Index Hint Register (`MFC_TLB_Index_Hint`).
- MFC Data-Storage Interrupt Status Register (`MFC_DSISR`).
- MFC TLB Index Register (`MFC_TLB_Index`).
- MFC TLB Real Page Number Register (`MFC_TLB_RPN`).
- MFC TLB Virtual-Page Number Register (`MFC_TLB_VPN`).
- MFC Data Address Register (`MFC_DAR`).

Descriptions of these registers are given in the following sections. See the *Cell Broadband Engine Registers* document for complete descriptions of the implemented portions of the MFC TLB MMIO registers defined in the *Cell Broadband Engine Architecture.* Writes to invalid register-offset addresses in the assigned SMM range are ignored. Reads to invalid addresses return ‘0’ data. The same applies to non-implemented bits.

The Synergistic Bus Interface (SBI) unit retrieves all register requests from the EIB and formats the requests for the MFC. All valid SMM requests are formatted for 64-bit data. The `MFC_DSISR` register can be addressed as a 32-bit register, but the SBI converts it to a 64-bit format.

SMM_HID Register

The SMM Hardware Implementation Dependent Register (`SMM_HID`) contains configuration bits that control many essential functions of the SMM. This register should be written before the SMM performs any translations and before the arrays are loaded with data.

The `Pg_Sz_Dcd` bits are the most important field in this register. These bits, along with the `LP` bit in the `SLB_VSID` register, determine the supported large-page sizes for each array entry when the L bit in `SLB_VSID` is '1', enabling large-page mode. The large-page (`LP`) sizes are determined during a write to the `SLB_VSID` register. See *Table 4-13* for large-page size decoding. By default, the `SMM_HID` register is cleared to ‘0’, so 16-MB pages are decoded. There are two sets of 2-bit HID values to support the two concurrent large-page sizes. This portion of the `SMM_HID` register should be written before any SMM activity occurs, and it can only change with a context switch.

Table 4-13. SMM_HID Page-Size Decoding

L	SLB[LP] or TLBIE[LS]	SMM_HID[0:1]	SMM_HID[2:3]	Page Size
0	—	—	—	4 KB
1	0	00	—	16 MB
1	0	01	—	1 MB
1	0	10	—	64 KB
1	0	11	—	Reserved
1	1	—	00	16 MB
1	1	—	01	1 MB
1	1	—	10	64 KB
1	1	—	11	Reserved

Table 4-13 is applicable for `MFC_TLB_Invalidate_Entry` purposes as well. The `MFC_TLB_Invalidate_Entry[L, LS]` bits are used with `SMM_HID[0:3]` to select an `MFC_TLB_Index` for invalidation. See the MFC TLB Invalidate Entry MMIO register in *Cell Broadband Engine Registers* for more information. SMM page sizes must agree with PPE page sizes for the same page table.

Parity bits in the SLB are generated during a write to the `SLB_VSID` register. Parity-generation can be disabled by setting `SMM_HID[SPGEN_Dsbl]` to ‘1’. The parity bits are checked during address translation and reads from the `SLB_VSID` register. Parity-checking can be disabled by setting `SMM_HID[SPCHK_Dsbl]` to ‘1’.

Similarly, parity bits are generated for the TLB entries during writes to the TLB and upon entry reloads by the table-lookup mechanism. TLB parity-generation can be disabled by setting `SMM_HID[TPGEN_Dsbl]` to ‘1’. Parity-checking occurs during address translation and reads from the `MFC_TLB_VPN` or `MFC_TLB_RPN` registers. The checking can be disabled by setting `SMM_HID[TPCHK_Dsbl]` to ‘1’. Enabling of parity-generation and disabling parity-checking causes data error (DERR) interrupts; see *Table 4-15 Translation, MMIO, and Address Match-Generated Exceptions* on page 117 for more about DERR interrupts.

RMT functionality in the SMM can be disabled by setting SMM_HID[RMT_Dsbl] to ‘1’.

With SMM_HID[Stall_DMA_Issue] equal to '0' (the default setting), a pending TLBIE condition stalls the DMAC from issuing new requests to the Atomic and SBI units. When SMM_HID[Stall_DMA_Issue] is set to '1', a pending TLBIE condition does not cause the DMAC to stall. With SMM_HID[ATO_TLBIE_Comp] equal to '0' (the default setting) and one of three livelock conditions (RC, castout, or PTE fetch) detected, the Atomic Unit does not force a pending TLBIE to complete. When SMM_HID[ATO_TLBIE_Comp] is set to '1', a detected livelock condition forces a TLBIE completion.

MFC_SDR Register

The MFC Storage Description Register (MFC_SDR) contains the page-table origin and size. The function of this register is identical to that of the PPE_SDR1 register.

MFC_TLB_Index_Hint Register

The function of the MFC_TLB_Index_Hint register is identical to that of the PPE_TLB_Index_Hint register, except that the SMM version has only one thread and implements a 6-bit index in bits[53:59].

MFC_TLB_Index, MFC_TLB_RPN, and MFC_TLB_VPN Registers

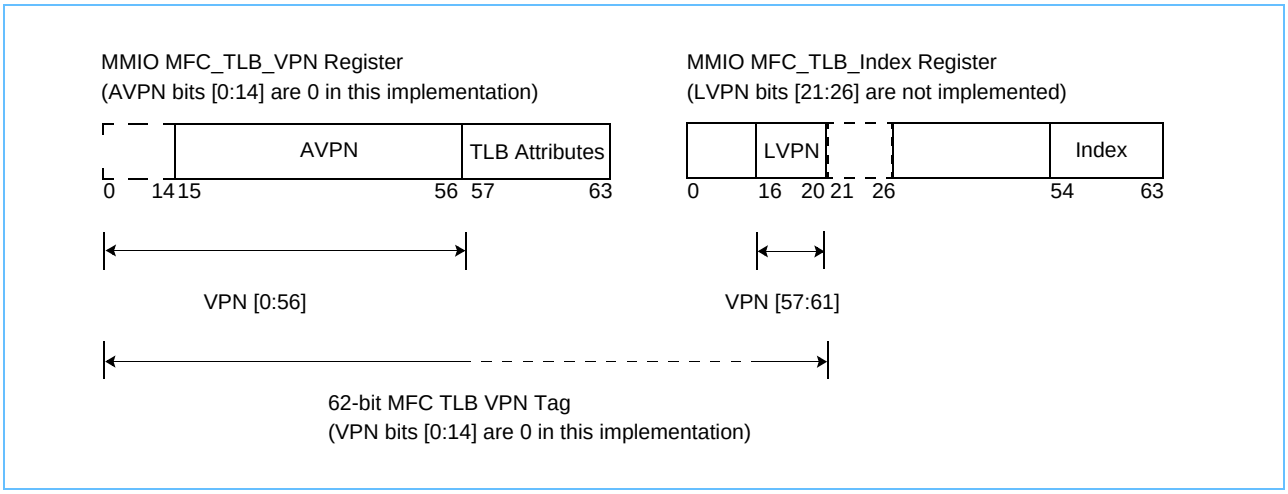
In order to write a new entry into the TLB, software must write the following registers, in the following order:

- MFC TLB Index (MFC_TLB_Index).
- MFC TLB Real-Page Number (MFC_TLB_RPN).
- MFC TLB Virtual-Page Number (MFC_TLB_VPN).

After this is done, hardware writes the entire TLB entry. Because an MMIO write to the MFC_TLB_VPN register sets off an automatic hardware write to the MFC_TLB_VPN and MFC_TLB_RPN register fields, whether the Index and RPN were previously written or not, the RPN data must be written after the index, as in the above order.

The complete MFC TLB VPN tag is created by concatenating the MFC_TLB_VPN[AVPN] and MFC_TLB_Index[LVPN] fields, as shown in *Figure 4-8*. This is the complete tag stored in the TLB during MMIO writes and used for comparison with the VA during TLB lookup.

Figure 4-8. MFC TLB VPN Tag



For TLB reads, the Index should be written first, as with the writes, but only bits [54:63]. MMIO VPN reads must occur in two separate MMIO reads. Reading the `MFC_TLB_VPN` register returns the register data just as it was written, and it also updates the `LVPN` field in the `MFC_TLB_Index` register. A separate `MFC_TLB_Index` register read must be done to retrieve the `LVPN` data. `MFC_TLB_RPN` register reads can occur independently and use the Index field of the `MFC_TLB_Index` register to select the entry to read.

The `W` bit of the `MFC_TLB_RPN` register is not implemented by the CBE processor. The R and M bits are not retained in the TLB array, but they are returned as ‘1’ during an MMIO read of the `MFC_TLB_RPN` register.

MFC_TLB_Invalidate_Entry Register

Writes to the `MFC_TLB_Invalidate_Entry` register invalidate all four ways within a TLB congruence class. To invalidate the entire array, this MMIO request must be executed 64 times for each index position.

To invalidate an entry, a 6-bit index must be selected from the `VPN` and `LP` fields. The correct range to select is determined from the `L` and `LS` bits in conjunction with the `SMM_HID[0:3]` bits. If `L` equals ‘0’, then the small-page (4-KB) index range is selected. When `L` equals ‘1’, the page size is determined as shown in *Table 4-14* on page 114.

Table 4-14. MFC_TLB_Invalidate_Entry Index Selection

Page Size	MFC_TLB_Invalidate_Entry Index Range
4 KB	MFC_TLB_Invalidate_Entry[56:61]
64 KB	MFC_TLB_Invalidate_Entry[52:57]
1 MB	MFC_TLB_Invalidate_Entry[48:53]
16 MB	MFC_TLB_Invalidate_Entry[44:49]

The `MFC_TLB_Invalidate_Entry` data is actually written into the `MFC_TLB_VPN` register because these two registers share VPN format data. There is no conflict in using the `MFC_TLB_VPN` register for both functions. The `MFC_TLB_Invalidate_Entry` register is write-only. The `MFC_TLB_VPN` register reads data retrieved from the array and not the `MFC_TLB_VPN` register. Reads from the `MFC_TLB_Invalidate_Entry` MMIO register return ‘0’ data.

MFC_TLB_Invalidate_All Register

The `MFC_TLB_Invalidate_All` register defined by the *Cell Broadband Engine Architecture* is not implemented in the CBE processor.

MFC_ACCR Register

The MFC Address Compare Control Register (`MFC_ACCR`) implements a Data Address Compare (DAC) mechanism that allows the detection of a DMAC access to either a virtual page with the Address Compare (AC) bit in the PTE set to ‘1’, or to a range of addresses within the LS. See the *PowerPC Operating Environment Architecture, Book III* for the DAC error-detection algorithm.

MFC_DSISR Register

The MFC Data-Storage Interrupt Status Register (`MFC_DSISR`) contains status bits relating to the last Data-Storage Interrupt (DSI) generated by the SMM. As DSIs are generated by the SMM, the status bits are written to this register in the middle of a translation request. If software attempts to write this register at the same time that the SMM is writing the register, the SMM hardware data overrides the software-written data (this makes the interrupt information available for software debug).

The `MFC_DSISR` register is used in this way, whether the SMM is operating in virtual addressing mode or real addressing mode. The SMM only writes to this register when a DSI is asserted. The register is locked until the Restart bit, `MFC_CNTL[R]`, is set to ‘1’. The register is never cleared; it is only overwritten with new DSI information.

MFC_DAR Register

The MFC Data Address Register (`MFC_DAR`) contains the 64-bit EA from a DMA command. The `MFC_DAR` register is writable both by the SMM hardware and by software. During translation, the SMM stores the EA in this register for every DMA command until a potential error is discovered. Then, the register becomes locked until the potential error passes or a previously reported error is cleared by setting the Restart bit, `MFC_CNTL[R]`, to ‘1’.

If SMM hardware and software attempt to write the `MFC_DAR` register simultaneously, the SMM hardware data overrides the software-written data (this makes the interrupt information available for software debug). This register is used for both virtual and real addressing modes. The EA is stored in this register for all SMM errors and interrupts, except parity errors on MMIO array reads.

4.3.5.4 *Storage Coherency*

The TLB maintains storage coherency by means of snooped TLBIE requests. The TLBIE requests are broadcast over the EIB and collected by the SBI unit. The SBI unit checks the Logical Partition ID (`LPID`) for the request to see if the TLBIE is targeted for its partition. If so, the request is sent to the Atomic Unit in the MFC. The Atomic Unit generates the TLBIE request for the SMM.

4.3.5.5 *TLB After POR*

All TLB entries and SLB entries are invalidated after Power-On Reset (POR).

4.3.6 Real Addressing Mode

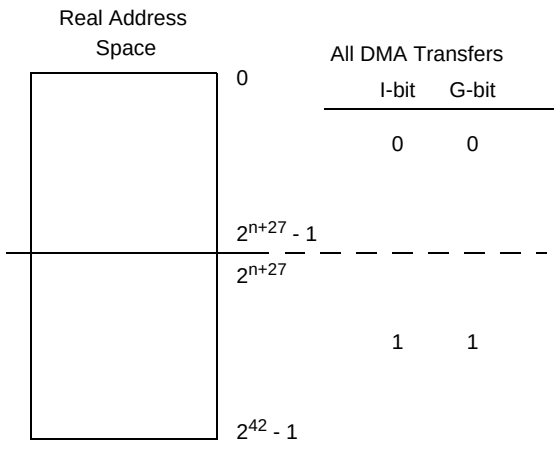
If PPE privileged software has set `MFC_SR1[R]` = ‘0’, the main-storage side of MFC accesses operates in real-addressing mode. In this mode, EAs provided to the MFC by programs are used as RAs. However, unlike the PPE’s real-addressing mode, described in *Section 4.2.8* on page 99, the SPE’s real-addressing mode does not use the RMOR or HRMOR base-address registers, or the `MSR[HV]` and `LPCR[LPES]` fields and the `RMLR` register.

The MFC supports a Real-Mode Address Boundary Facility, specified by the 4-bit `MFC_RMAB[RMB]` register field. The RMB field is used to control caching-inhibited (I) and guarded (G) storage-control attributes when the MFC is running in real mode. The field sets a boundary, illustrated in *Figure 4-9*, between storage that is considered well-behaved and cacheable and storage that is not. The granularity of this boundary is 256 MB.

Specifically, when in real-addressing mode, if `MFC_RMAB[RMB]` is set to a value of ‘n’ (for $1 \leq n \leq 15$), only those accesses within the first 2^{n+27} bytes of the real-address space are considered cacheable and well-behaved (that is, load-load or load-store ordering can be assumed). All accesses outside the first 2^{n+27} bytes are considered neither cacheable nor well-behaved. If `MFC_RMAB[RMB]` = x‘0’, no accesses are considered well-behaved and cacheable.

The RMB field has no effect when the MFC is running with translation on (`MFC_SR1[R]` = ‘1’). PPE privileged software must suspend all MFC operations before modifying the contents of `MFC_RMAB[RMB]` field.

Figure 4-9. Real-Mode Storage Boundary



4.3.7 Exception Handling and Storage Protection

The SMM checks for storage-protection errors during both virtual-mode and real-mode address translation, although real-mode checking is very limited. The DMAC also checks for LS-address compares. *Table 4-15* describes translation, address-compare, and parity errors detected and reported by the SMM and DMAC. These exceptions are reported in the following MMIO registers:

- Class 1 Interrupt Status Register (`INT_Stat_class1`).
- MFC Data-Storage Interrupt Status Register (`MFC_DSISR`).
- MFC Fault Isolation Register (`MFC_FIR`).

See the *Cell Broadband Engine Registers* document for a description of the seven Fault Isolation Registers (FIRs), including MFC Fault Isolation Register (`MFC_FIR`), MFC Fault Isolation Register Error Mask (`MFC_FIR_Err`), and MFC Fault Isolation Register Checkstop Enable (`MFC_FIR_ChkStpEnbl`) registers.

Table 4-15. Translation, MMIO, and Address Match-Generated Exceptions (Page 1 of 2)

Exception Name	INT_Stat_class1 Register			MFC_DSISR Register	DMAC Suspend	Data Error (DERR)	MFC_FIR Register
	LP or LG Bits[60:61]	MF (DSI) Bit[62]	SF Bit[63]				
SLB Segment Fault			X				
Atomic access to caching-inhibited page when address relocation is enabled		X		A bit[37]	X		
Atomic access to caching-inhibited page when address relocation is disabled (Real Mode)		X		A bit[37]	X		

Table 4-15. Translation, MMIO, and Address Match-Generated Exceptions (Page 2 of 2)

Exception Name	INT_Stat_class1 Register			MFC_DSISR Register	DMAC Suspend	Data Error (DERR)	MFC_FIR Register
	LP or LG Bits[60:61]	MF (DSI) Bit[62]	SF Bit[63]				
Page protection violation (PP)		X		P bit[36]			
Local-storage address compare (reported by DMAC and not reported as SMM exception)	X				X		
Data address compare (DAC)		X		C bit[41]	X		
TLB page fault (hardware or software)		X		M bit[33]			
Parity error detected during SLB/TLB translation operation					X		SLB or TLB bits [43:44}
Parity error detected during MMIO read to SLB or TLB array (only reported through DERR bit attached to the MMIO data returned to the requester)						X	

4.3.7.1 *Address-Translation Exceptions*

For all translation exceptions reported by the SMM, except parity errors detected on SLB or TLB MMIO reads, the 64-bit EA is saved to the `MFC_DAR` register. If a DSI exception occurs, information about which kind of DSI error and other related information is written into the `MFC_DSISR` register, and the MFC Command Queue Error ID Register (`MFC_CMDQ_Err_ID`) entry index is saved to the MFC Data-Storage Interrupt Pointer Register (`MFC_DSIPR`). The `MFC_DAR`, `MFC_DSIPR`, and `MFC_DSISR` registers are locked to the value of the current outstanding exception or miss.

The SMM supports only one outstanding translation miss or exception, excluding MMIO-read parity errors. The SMM has the ability to serve hits under one outstanding translation miss or exception. After the first outstanding miss or exception is detected by the SMM, any subsequent exception or miss of an MFC-command translation is reported back to the DMAC as a miss. The DMAC sets the dependency bit associated with the MFC command and stalls the command until the dependency bit is cleared.

When the outstanding miss or exception condition is cleared, all 24 dependency bits are cleared (16 for the MFC SPU Command Queue and 8 for the MFC Proxy Command Queue), and the DMAC resends the oldest MFC command to the SMM for translation. Exceptions are cleared by setting the Restart bit, `MFC_CNTL[R]`, to ‘1’. Setting `MFC_CNTL[R]` to ‘1’ also unlocks the `MFC_DAR` and `MFC_DSISR` registers and allows the SMM to report a new exception. The `MFC_CNTL[Sc]` bit should be cleared to ‘0’ to unsuspend the MFC command queues only after all exception conditions have been cleared. The `MFC_DSIPR` register is unlocked by reading the register or purging the MFC command queues.

4.3.7.2 *Address-Compare Exceptions*

LS-address-compare exceptions occur when the LS is accessed within the address range specified in the MFC Local Store Address Compare Register (`MFC_LSACR`). The DMAC writes the MFC Local Store Compare Results Register (`MFC_LSCRR`) with the LS address that met the compare conditions set in the `MFC_LSACR` register, along with the MFC command-queue index and the DMA command type. This exception causes the DMAC to suspend. To clear the error, the `LP` or `LG` bits in the `INT_Stat_class1` register should be cleared to '0', and the `MFC_CNTL[Sc]` bit should be cleared to '0' to unsuspend the MFC command queues. See the MFC Local Store Compare Results (`MFC_LSCRR`) MMIO register and the MFC Local Store Address Compare (`MFC_LSACR`) MMIO register in *Cell Broadband Engine Registers* for more information.

4.3.7.3 *Segment Exceptions*

SLB segment faults occur when there is no matching ESID in the SLB for the EA in the translation request, or the Valid bit is not set in the SLB for the matching ESID entry. When this fault occurs, the `MFC_DAR` register contains the 64-bit EA, as described above. The `MFC_DAR` register and the SLB can be read to determine which entry to replace in the SLB. Then, the entry should be invalidated with a write to the `SLB_Invalidate_Entry` or `SLB_Invalidate_All` register, followed by writes to the `SLB_Index`, `SLB_ESID`, and `SLB_VSID` registers to put the proper entry into the SLB. The `INT_Stat_class1[SF]` should then be cleared to '0', and the `MFC_CNTL[R]` bit should be set to '1' to resume normal SMM and DMAC operation.

4.3.7.4 *Data-Storage Exceptions*

All DSI exceptions are defined in the *PowerPC Operating Environment Architecture, Book III*. When these exceptions occur, the SMM writes the 64-bit EA to the `MFC_DAR` register and records the DSI exception type in the `MFC_DSISR` register, as shown in *Table 4-15* on page 117. That table also shows which DSI exceptions cause the DMAC to suspend. `MFC_DSISR[S]` is also set to '1' if the DMAC translation request was a **put[rlfs]**, **putll[u]c**, or **sdcrz** operation. The SMM reports the DSI to the SBI in order to set the `INT_Stat_class1[MF]` bit to '1'. Software can read the `MFC_DAR` and `MFC_DSISR` registers and the SLB and TLB arrays to determine the type of DSI fault and the method for fixing it. After the error condition has been fixed, the `INT_Stat_class1[MF]` bit should be cleared to '0', and the `MFC_CNTL[R]` bit should be set to '1'.

4.3.7.5 *Parity Exceptions*

Software may be able to recover from an SLB or TLB parity errors by writing over the faulty entry. Parity errors detected during address translation set bits in the `MFC_FIR` register as shown in *Table 4-15* on page 117. Only parity errors detected on an address translation cause the `MFC_DAR` to be locked with the EA value captured in it. After the error has been fixed, the `MFC_FIR` bits should be cleared and the `MFC_DSIPR` register written to restart the SMM and unsuspend the DMAC. Parity errors detected during MMIO reads on the SLB or TLB cause the DERR bit to be set on the EIB when the MMIO data is returned to the requester. The SMM can support an unlimited number of MMIO-read parity errors because they do not lock the SMM or DMAC interrupt registers or affect address translation.

5. Memory Map

Section 4 Virtual Storage Environment on page 79 describes how memory addresses are translated (mapped) between virtual-address, effective-address, and real-address forms. This section describes how real addresses are mapped to the real-address space, which includes physical memory, memory-mapped registers, and the SPE local store (LS) areas.

5.1 Introduction

Figure 5-1 on page 122 shows the memory map of the CBE processor's real-address space. The locations in the map are indicated by the sum of two values—an explicit real address added to a base real address. For example, "`x'1E 0000' + SPE3 BE_MMIO_Base`" means that the real address `x'1E 0000'` is added to the real address contained in the `SPE3 BE_MMIO_Base` base-address register. The region marked "Available to Software" can be used by software for any purpose if it is known to be populated with memory chips or external MMIO registers.

Although the *Cell Broadband Engine Architecture (CBEA)* defines only one base register (`BP_Base`) for relocating all of this address space, the CBE processor implements multiple base registers for this relocation, each associated with one of the main CBE logic groups—PPE, SPEs, Cell Broadband Engine Interface (BEI) Unit, Memory Interface Controller (MIC), and Pervasive Logic (PRV). The base-register values are initialized from the Configuration Ring during Power-On Reset (POR), as described in the *Cell Broadband Engine Hardware Initialization Guide*. System developers should use the same address for all eight SPE base addresses. *Table 5-2* on page 124 provides details about the allocated regions of memory.

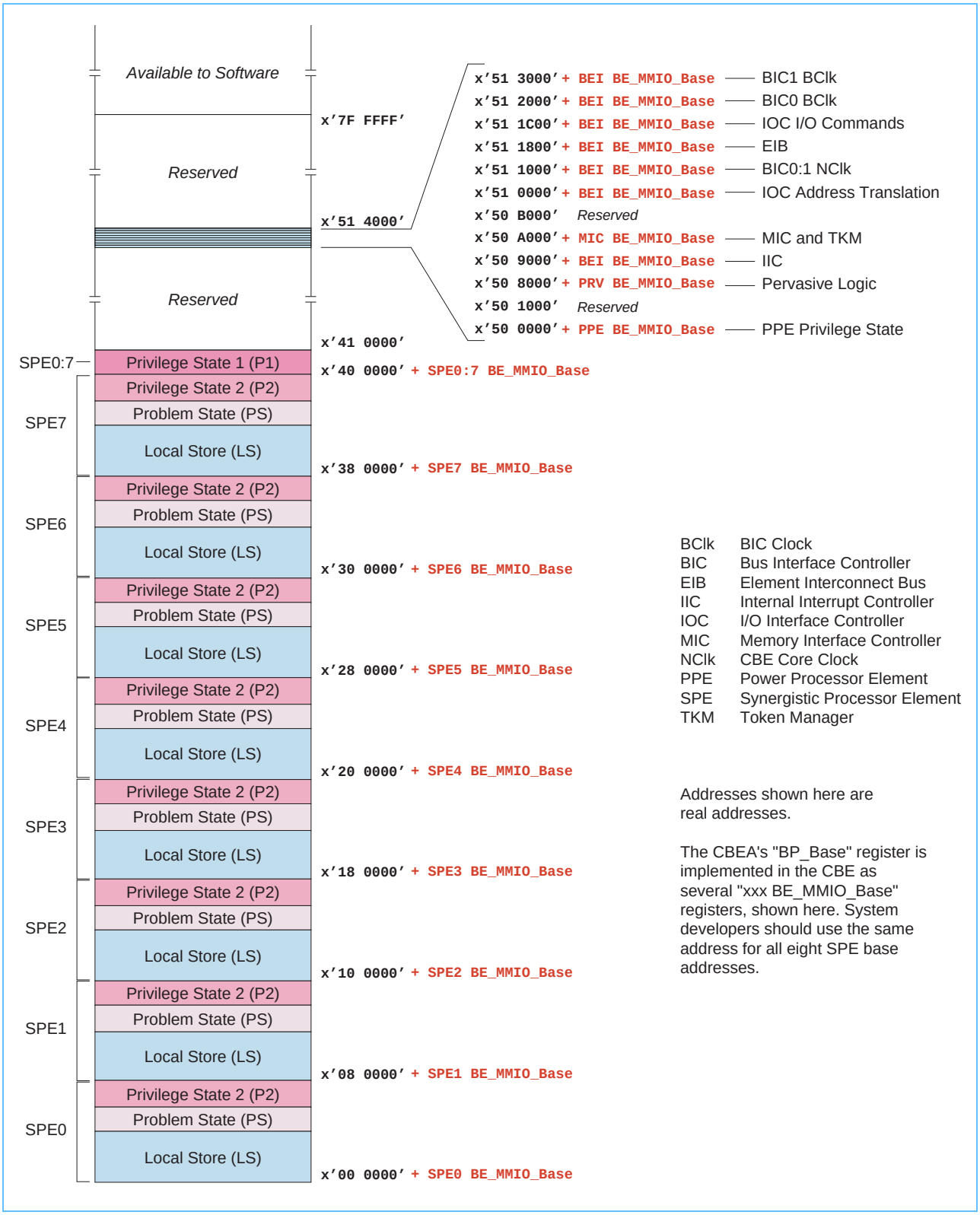
Areas of memory that are marked as *reserved* in *Figure 5-1* are not assigned to any functional unit and should not be read from or written to: doing so will cause serious errors in software.[9] Areas of memory that are not identified as allocated in *Figure 5-1*, and that are known to be populated with memory devices, are available to software for any purpose.

When accessing memory that is shared by the PPE, SPEs, or I/O devices, it may be necessary to synchronize storage accesses. The facilities for doing so are described in *Section 20 Shared-Storage Synchronization* on page 551. Loading cooperative PPE and SPE programs into CBE memory requires loading the PPE and SPE programs into distinct memory spaces. The procedures for doing so are described in *Section 14 Objects, Executables, and SPE Loading* on page 391. Additional information regarding stack frames and other software conventions for the use of memory are given in the *SPU Application Binary Interface Specification*.

The documentation for the CBE processor defines a "memory-mapped I/O (MMIO) register" as any internal or external register that is accessed through the main-storage space with load and store instructions, whether or not the register is associated with an I/O device.

9. See *Reserved Regions of Memory and Registers* on page 32 for further details.

Figure 5-1. CBE Memory Map



5.1.1 Configuration-Ring Initialization

Table 5-1 shows the memory-base registers for each of the main CBE logic groups. These base registers are initialized from the Configuration Ring during Power-On Reset (POR), as described in the *Cell Broadband Engine Hardware Initialization Guide*.

Table 5-1. Memory-Base Registers Loaded by Configuration Ring

Memory-Base Register	Logic Group	Number of Bits in Configuration-Ring Field	Section of Configuration Ring
SPE0 BE_MMIO_BASE SPE1 BE_MMIO_BASE SPE2 BE_MMIO_BASE SPE3 BE_MMIO_BASE SPE4 BE_MMIO_BASE SPE5 BE_MMIO_BASE SPE6 BE_MMIO_BASE SPE7 BE_MMIO_BASE	• SPE0:7 – Local Store (LS) – Problem State (PS) – Privilege State 2 (P2) – Privilege State 1 (P1)	19 for each SPE	MFC
PPE BE_MMIO_BASE	• PPE	30	PPSS
BEI BE_MMIO_BASE	• Cell Broadband Engine Interface (BEI) Unit: – Element Interconnect Bus (EIB) – Internal Interrupt Controller (IIC) – I/O Controller (IOC) I/O-Address Translation – I/O Controller (IOC) I/O Commands – Bus Interface Controller (BIC)	22	IOC
MIC BE_MMIO_BASE	• Memory Interface Controller (MIC)	30	MIC
PRV BE_MMIO_BASE	• Pervasive Logic (PRV)[1]	30	MIC

1. Pervasive Logic includes power management, thermal management, clock control, software-performance monitoring, trace analysis, and so forth.

The bit-fields for the base registers in the Configuration Ring correspond to the most-significant bits of the 42-bit real address implemented in the CBE processor. The most-significant 19 bits of all these Configuration-Ring fields should be set to the same value. If a Configuration-Ring field has more than 19 bits, the additional bits should be set to a value consistent with the settings in *Table 5-2* on page 124 for the starting address of that area of memory. Although each SPE has its own `BE_MMIO_Base` register field in the Configuration Ring, all such fields should be initialized with the same value.

5.1.2 Allocated Regions of Memory

Table 5-2 on page 124 provides details about the allocated regions of memory.

Table 5-2. CBE Memory Map (Page 1 of 2)

Base Register from Configuration Ring	Offset From Base Register		Area[1]		Size in Hex
	Start	End			
SPE0 BE_MMIO_Base	x‘00 0000’	x‘03 FFFF’	SPE0	Local Store	x‘4 0000’
	x‘04 0000’	x‘05 FFFF’		Problem State	x‘2 0000’
	x‘06 0000’	x‘07 FFFF’		Privilege 2 Area	x‘2 0000’
SPE1 BE_MMIO_Base	x‘08 0000’	x‘0B FFFF’	SPE1	Local Store	x‘4 0000’
	x‘0C 0000’	x‘0D FFFF’		Problem State	x‘2 0000’
	x‘0E 0000’	x‘0F FFFF’		Privilege 2 Area	x‘2 0000’
SPE2 BE_MMIO_Base	x‘10 0000’	x‘13 FFFF’	SPE2	Local Store	x‘4 0000’
	x‘14 0000’	x‘15 FFFF’		Problem State	x‘2 0000’
	x‘16 0000’	x‘17 FFFF’		Privilege 2 Area	x‘2 0000’
SPE3 BE_MMIO_Base	x‘18 0000’	x‘1B FFFF’	SPE3	Local Store	x‘4 0000’
	x‘1C 0000’	x‘1D FFFF’		Problem State	x‘2 0000’
	x‘1E 0000’	x‘1F FFFF’		Privilege 2 Area	x‘2 0000’
SPE4 BE_MMIO_Base	x‘20 0000’	x‘23 FFFF’	SPE4	Local Store	x‘4 0000’
	x‘24 0000’	x‘25 FFFF’		Problem State	x‘2 0000’
	x‘26 0000’	x‘27 FFFF’		Privilege 2 Area	x‘2 0000’
SPE5 BE_MMIO_Base	x‘28 0000’	x‘2B FFFF’	SPE5	Local Store	x‘4 0000’
	x‘2C 0000’	x‘2D FFFF’		Problem State	x‘2 0000’
	x‘2E 0000’	x‘2F FFFF’		Privilege 2 Area	x‘2 0000’
SPE6 BE_MMIO_Base	x‘30 0000’	x‘33FFFF’	SPE6	Local Store	x‘4 0000’
	x‘34 0000’	x‘35 FFFF’		Problem State	x‘2 0000’
	x‘36 0000’	x‘37 FFFF’		Privilege 2 Area	x‘2 0000’
SPE7 BE_MMIO_Base	x‘38 0000’	x‘3B FFFF’	SPE7	Local Store	x‘4 0000’
	x‘3C 0000’	x‘3D FFFF’		Problem State	x‘2 0000’
	x‘3E 0000’	x‘3F FFFF’		Privilege 2 Area	x‘2 0000’
SPE0 BE_MMIO_Base	x‘40 0000’	x‘40 1FFF’	SPE0	Privilege 1 Area	x‘2000’
SPE1 BE_MMIO_Base	x‘40 2000’	x‘40 3FFF’	SPE1		x‘2000’
SPE2 BE_MMIO_Base	x‘40 4000’	x‘40 5FFF’	SPE2		x‘2000’
SPE3 BE_MMIO_Base	x‘40 6000’	x‘40 7FFF’	SPE3		x‘2000’
SPE4 BE_MMIO_Base	x‘40 8000’	x‘40 9FFF’	SPE4		x‘2000’
SPE5 BE_MMIO_Base	x‘40 A000’	x‘40 BFFF’	SPE5		x‘2000’
SPE6 BE_MMIO_Base	x‘40 C000’	x‘40 DFFF’	SPE6		x‘2000’
SPE7 BE_MMIO_Base	x‘40 E000’	x‘40 FFFF’	SPE7		x‘2000’

BClk: BIC Clock.
BIC: Bus Interface Controller.
EIB: Element Interconnect Bus.
IIC: Internal Interrupt Controller.
IOC: I/O Interface Controller.
NClk: CBE Core Clock.
PRV: Pervasive Logic.

Table 5-2. CBE Memory Map (Page 2 of 2)

Base Register from Configuration Ring	Offset From Base Register		Area[1]		Size in Hex
	Start	End			
	x‘41 1000’	x‘4F FFFF’	*Reserved*[2]		
PPE BE_MMIO_Base	x‘50 0000’	x‘50 0FFF’	PPE	Privilege Area	x‘1000’
	x‘50 1000’	x‘50 7FFF’	*Reserved*[2]		
BEI BE_MMIO_Base	x‘50 8000’	x‘50 8FFF’	IIC		x‘1000’
PRV BE_MMIO_Base	x‘50 9000’	x‘50 93FF’	PRV	Trace Logic Array	x‘0400’
	x‘50 9400’	x‘50 97FF’		Performance Monitor	x‘0400’
	x‘50 9800’	x‘50 9BFF’		Thermal and Power Management	x‘0400’
	x‘50 9C00’	x‘50 9FFF’		Reliability, Availability, Serviceability (RAS)	x‘0400’
MIC BE_MMIO_Base	x‘50 A000’	x‘50 AFFF’	MIC and TKM		x‘1000’
	x‘50 B000’	x‘50 FFFF’	*Reserved*[2]		
BEI BE_MMIO_Base	x‘51 0000’	x‘51 0FFF’	IOC	I/O Address Translation	x‘1000’
	x‘51 1000’	x‘51 13FF’	BIC	BIC0 NClk	x‘0400’
	x‘51 1400’	x‘51 17FF’		BIC1 NClk	x‘0400’
	x‘51 1800’	x‘51 1BFF’	EIB		x‘0400’
	x‘51 1C00’	x‘51 1FFF’	IOC	I/O Commands	x‘0400’
	x‘51 2000’	x‘51 2FFF’	BIC	BIC0 BClk	x‘1000’
	x‘51 3000’	x‘51 3FFF’		BIC1 BClk	x‘1000’
	x‘51 4000’	x‘7F FFFF	*Reserved*[2]		
	x‘80 0000’	System Maximum	*Available to Software*[3]		

BClk: BIC Clock.
BIC: Bus Interface Controller.
EIB: Element Interconnect Bus.
IIC: Internal Interrupt Controller.
IOC: I/O Interface Controller.
NClk: CBE Core Clock.
PRV: Pervasive Logic.

1. See *Section 11.3.1 Access Privilege* on page 340 for definitions of SPE Privilege 1 and Privilege 2.
2. Areas of memory that are marked as reserved are not assigned to any functional unit and should not be read from or written to: doing so will cause serious errors in software.
3. Areas of memory that are not identified as allocated in *Figure 5-1*, and that are known to be populated with memory chips or external MMIO registers, are available to software for any purpose.

5.1.3 Reserved Regions of Memory

Regions within the memory map that are marked as *reserved* in *Table 5-2* operate as follows: writes have no effect, and reads return all ones. Reserved areas are not assigned to any unit and should not be read from or written to: doing so will cause serious errors in software as follows. For reads or writes generated from the SPE to unassigned reserved spaces, at least one of the following `MFC_FIR[46,53,56,58,61]` bits will be set and in most cases will cause a Checkstop. For reads or writes generated from the PPE to unassigned reserved spaces, at least one of the

CIU_FIR[7,8] will be set and a Checkstop will occur. For reads or writes generated from the IOC to unassigned reserved spaces, the IOC will respond back to the IOIF device that sourced the address request with a ERR (error) response. No IOC FIR bits are set.

5.1.4 The Guarded Attribute

If some areas of real memory are not fully populated with memory devices, privileged software may wish to give the pages that map these areas the guarded attribute, as described in *Section 4 Virtual Storage Environment* on page 79. The guarded attribute can protect the system from undesired accesses caused by out-of-order load operations or instruction prefetches that could lead to the generation of a Machine Check exception. Also, the guarded attribute can be used to prevent out-of-order (speculative) load operations or prefetches from occurring to I/O devices that produce undesired results when accessed in this way. However, the guarded attribute can result in lower performance than would be the case with unguarded storage.

5.2 PPE Memory Map

5.2.1 PPE Memory-Mapped Registers

Figure 5-1 on page 122 and *Table 5-1* on page 123 show the PPE memory area and its offset from the value in the PPE BE_MMIO_Base register that is filled by the Configuration Ring at POR. All locations within this area that are not populated as described in this chapter are reserved.

Table 5-3 lists the groups of PPE MMIO registers in this memory area. All registers in this area can only be accessed by privileged software. For details about the registers in these groups, see the *Cell Broadband Engine Registers* document and the *PowerPC Operating Environment Architecture, Book III*.

Table 5-3. PPE Privileged Memory-Mapped Register Groups

x'50 0000' + Offset From Base Register		Register Group
Start	End	
x'0310'	x'0886'	Level 2 (L2) MMIO Registers
x'0900'	x'0976'	Core Interface Unit (CIU) MMIO Registers
x'0A48'	x'0A68'	Noncacheable Unit (NCU) MMIO Registers
x'0B00'	x'0B78'	Bus Interface Unit (BIU) MMIO Registers

5.2.2 Predefined Real-Address Locations

There are four areas of the PPE real-address space[10] that have predefined uses, as shown in *Table 5-4*. The first 256 bytes are reserved for operating-system use. The remainder of the 4-KB page beginning at address x'0000' is used for exception vectors. The two contiguous 4-KB pages beginning at real address x'1000' are reserved for implementation-specific purposes. A contig-

10. In the real addressing mode for the PPE, accesses to main storage are performed in a manner that depends on the contents of the MSR[HV] bit, the LPCR[LPES] field, and the HRMOR, RMLR, and RMOR registers. For details, see *Section 4.2.8 Real Addressing Mode* on page 99, *Section 11 Logical Partitions and a Hypervisor* on page 327, and *PowerPC Operating Environment Architecture, Book III*.

uous sequence of real pages beginning at the real address specified by Storage Description Register 1 (SDR1) contains the page table. See the *PowerPC Operating Environment Architecture, Book III* for information about the SDR1 register.

Table 5-4. Predefined Real-Address Locations

Real Address		Predefined Use
Start	End	
x'0000'	x'00FF'	Operating system
x'0100'	x'0FFF'	Exception vectors[1]
x'1000'	x'2FFF'	Implementation-specific use
Software-specified in Storage Description Register 1 (SDR1)		Page table

1. See *Section 9 PPE Interrupts* on page 237 for the assignment of the exception-vector offsets.

5.3 SPE Memory Map

Each SPE is allocated an area of memory for the following purposes:

- LS, if aliased to main-storage space (see *Section 5.3.1* on page 128).
- Problem State MMIO Registers.
- Privilege State 2 MMIO Registers.
- Privilege State 1 MMIO Registers.

Figure 1-2 on page 40 shows the relationship between an SPE's LS and main storage. SPE software accesses its LS directly, by fetching instructions and loading and storing data using an LS address (LSA). SPE software has no direct access to main storage; instead, it must use the MFC DMA facilities to move data between the LS and the effective-address (EA) space.

Access, by an SPE, to its channels that are associated with its MMIO registers can be done through the use of read channel (**rdch**) and write channel (**wrch**) instructions. See *Section 17 SPE Channel and Related MMIO Interface* on page 441 and *Section 19 DMA Transfers and Inter-Processor Communication* on page 505 for details.

Access to an SPE's MMIO registers by the PPE and other SPEs and devices that can generate EAs is done through the EA space. As indicated above, the SPE's MMIO registers are divided into three groups, based on privilege:

- *Privilege State 1 (most privileged)*—Used by a hypervisor (*Section 11* on page 327) to manage an SPE on behalf of a logical partition or, if there is no hypervisor, used by the operating system.
- *Privilege State 2*—Used by the operating system in a logical partition to manage the SPE within the partition or, if there is no hypervisor, used by the operating system.
- *Problem State (Privilege 3, least privileged*)—Used by Problem State (user, or application) software, if direct access to the SPE from user space is supported by the operating system.

5.3.1 SPE Local-Store Memory Map

The LS area for each SPE is 256-KB in size. If these LS areas are aliased into the EA space, they begin at the RA offsets shown in *Table 5-2* on page 124.

5.3.1.1 *LS Aliasing to Main Storage*

Privileged software on the PPE can alias an SPE's LS address space to the main-storage EA space. The aliasing is done by setting the LS Real-Address Decode (D) bit of the MFC State Register One (`MFC_SR1`) to '1'. This aliasing allows the PPE, other SPEs, and external I/O devices to physically address an LS through a translation method supported by the PPE or SPE, or, for I/O devices, through a translation method supported by the EIB (*Section 7.4* on page 176).

When aliasing is implemented, each LS address (LSA) is assigned a real address (RA) within the effective-address (EA) space. Privileged software can, for example, map an LSA space into the EA space of an application to allow DMA transfers between the LS of one SPU and the LS of another SPU. Data transfers that use the LS area aliased in main storage should do so as caching-inhibited storage, because these accesses are not coherent with the SPU LS accesses (that is, SPU load, store, instruction fetch) in its LS domain.

Each SPE has its own `MFC_SR1` register, located in the Privilege 1 memory-map area (*Figure 5-1* on page 122) for that SPE. When an SPE's `MFC_SR1[D]` bit is set to '1', the MFC associated with that SPE decodes RAs into LSAs.

As an alternative to aliasing, privileged software can use the `MFC_SR1[D]` bit to prevent the LS from being addressed using RAs. This can be done to facilitate a more isolated SPU program environment.

The value of the `MFC_SR1[D]` bit has no effect on LS access by MFC DMA commands or on SPU loads and stores from LS, both of which use LSAs to access the LS.

5.3.1.2 *Coherence*

As mentioned above, all SPE memory area is architecturally defined as noncoherent. Privileged software on the PPE should access aliased pages of LS through the main-storage domain. The pages must either have the caching-inhibited attribute (*Section 4 Virtual Storage Environment* on page 79), or software must explicitly manage the coherency of LS with other system caches.

5.3.1.3 *Data-Transfer Performance*

Since aliased LS must be treated as non-cacheable, transferring a large amount of data using the PPE load and store instructions can result in poor performance. Data transfers between the LS domain and the main-storage domain should use the MFC DMA commands to avoid stalls.

5.3.1.4 *LS-Access Errors*

MFC commands that access an EA range that maps to its own LS can produce an error or unpredictable results. If the two address ranges of a DMA transfer (translated EA and LSA) overlap and the source is a lower address than the destination, the DMA transfer results in the corruption of the source data. Address overlap is not detectable and does not generate an exception. Therefore, it is software's responsibility to avoid an unintended overlap.

5.3.2 SPE Memory-Mapped Registers

Figure 5-1 on page 122 and *Table 5-1* on page 123 show the SPE memory areas and their offsets from the value in the `SPE BE_MMIO_Base` registers that are filled by the Configuration Ring at POR. All locations within this area that are not populated as described in this chapter are reserved.

Table 5-5 lists the groups of SPE problem-state registers, using SPE0 as an example for address offsets (see *Table 5-2* on page 124 for details). *Table 5-6* lists the groups of SPE privilege state 2 registers, using SPE0 as an example. *Table 5-6* lists the groups of SPE privilege state 1 registers, using SPE0 as an example. For details about the registers in these groups, see the *Cell Broadband Engine Registers* document.

Table 5-5. SPE Problem-State Memory-Mapped Register Groups

Offset From Base Register + x'4000'		Register Group
Start	End	
x'0000'	x'2FFF'	SPE Multisource Sync Register.
x'3004'	x'3017'	MFC Command Parameter Registers.
x'3104'[1]	x'322F'	MFC Command Queue Control Registers.
x'4004'	x'4038'	SPU Control Registers.
x'1 400C'	x'1 FFFF'	Signal Notification Registers.

1. This address is shared by multiple registers. See *Table 17-2* on page 443 for details.

Table 5-6. SPE Privilege-State-2 Memory-Mapped Register Groups

Offset From Base Register + x'6000'		Register Group
Start	End	
x'1100'	x'1FFF'	SLB Management Registers.
x'2000'	x'2FFF'	Context Save and Restore Register.
x'3000'	x'3FFF'	MFC Control Register.
x'4000'	x'4008'	Interrupt Mailbox Register.
x'4040'	x'4FFF'	SPU Control Registers.
x'5000'	x'1 FFFF'	Context Save and Restore Registers.

Table 5-7. SPE Privilege-State-1 Memory-Mapped Register Groups (Page 1 of 2)

Offset From Base Register + x'40 0000'		Register Group
Start	End	
x'0000'	x'00FF'	MFC Registers.
x'0100'	x'01FF'	Interrupt Registers.
x'0200'	x'028F'	Atomic Unit Control Registers.
x'0290'	x'03B7'	Fault Isolation Registers.
x'03B8'	x'03C7'	Miscellaneous Registers.
x'03C8'	x'03FF'	Performance Monitor Registers.

Table 5-7. SPE Privilege-State-1 Memory-Mapped Register Groups (Page 2 of 2)

Offset From Base Register + x'40 0000'		Register Group
Start	End	
x'0400'	x'0547'	MFC TLB Management Registers.
x'0580'	x'0587'	Memory Management Register.
x'0600'	x'06FF'	MFC Status and Control Registers.
x'0710'	x'07FF'	Replacement Management Table (RMT) Registers.
x'0800'	x'087F'	MFC Command Data-Storage Interrupt Registers.
x'0880'	x'08FF'	DMAC Unit Performance Monitor Control Register.
x'0900'	x'0BFF'	Real-Mode Support Registers.
x'0C00'	x'0FFF'	MFC Command Error Register.
x'1000'	x'101F'	SPU ECC and Error Mask Registers.
x'1028'	x'13FF'	SPU Performance Monitor Control Registers.
x'1400'	x'1FFF'	Performance Monitor Register.

5.4 BEI Memory-Mapped Registers

The Cell Broadband Engine Interface (BEI) Unit memory space contains registers for the following functional units:

- Element Interconnect Bus (EIB).
- Internal Interrupt Controller (IIC).
- I/O Controller (IOC) I/O-Address Translation.
- I/O Controller (IOC) I/O Commands.
- Bus Interface Controller (BIC).

For details about these registers, see the *Cell Broadband Engine Registers* document. For details about I/O functions performed by these units, see *Section 7 I/O Architecture* on page 161.

5.4.1 I/O

The IOC Command (IOCmd) unit handles commands that originate from the EIB and go to IOIF0 or IOIF1. The IOCmd unit also acts as a proxy for commands that originate from IOIF0 or IOIF1 and go to the EIB. Commands that originate from IOIF0 or IOIF1 are sent with an I/O address. If I/O-address translation is enabled by setting `IOC_IOCmd_Cfg[16]` = '1', the IOCmd unit provides buffering while the I/O addresses are being translated into RAs.

The IOCmd unit accepts commands from the EIB for which the addresses match the RA ranges specified for each IOIF, using the following registers:

- IOC Base Address Register 0 (`IOC_BaseAddr0`).
- IOC Base Address Register 1 (`IOC_BaseAddr1`).
- IOC Base Address Mask Register 0 (`IOC_BaseAddrMask0`).
- IOC Base Address Mask Register 1 (`IOC_BaseAddrMask1`).

These IOC base-address registers are filled, in part, by Configuration Ring fields at POR. Specifically, hardware copies the following Configuration-Ring fields into the IOC base-address registers:

- Configuration-Ring IOIF0 Base Address and Replacement field is copied to `IOC_BaseAddr0[22:32,53:63]`.
- Configuration-Ring IOIF1 Base Address and Replacement field is copied to `IOC_BaseAddr1[22:32,53:63]`.
- Configuration-Ring IOIF0 Base Address Mask field is copied to `IOC_BaseAddrMask0[0,22:32]`.
- Configuration-Ring IOIF1 Base Address Mask field is copied to `IOC_BaseAddrMask1[0,22:32]`.

The IOC base-address registers are writable by privileged software, so that the I/O portion of the memory map can be relocated after POR.

6. Cache Management

The CBE processor has several types of caches. Primary among them are the PPE's Level-1 (L1) instruction and data caches, and its unified Level-2 (L2) cache. The contents of these caches are maintained coherently with the contents of main storage on a 128-byte cache-line basis, and the *PowerPC Architecture* cache-control instructions support user-level operations on cache lines. In addition to the L1 and L2 caches, the PPE and the SPEs have other caches, queues, and arrays that enhance performance and can be controlled by software.

The basic PPE cache architecture is described in the *PowerPC Virtual Environment Architecture, Book II*, *PowerPC Operating Environment Architecture, Book III*, and *PowerPC Microprocessor Family: The Programming Environments for 64- and 32-Bit Microprocessors*. Features of the SPE caches are described later in this chapter. For detailed descriptions of the caches used in memory management, see *Section 4 Virtual Storage Environment* on page 79.

6.1 PPE Caches

In addition to the L1 and L2 caches, the PPE also has other caches, queues, and arrays that support memory management and act as predecessors and extensions to the L1 and L2 caches. For example, the PPE contains store queues for holding data that is in flight between the load and store units and the L2 cache, and castout queues for holding modified data that has been pushed out of the L2 cache. All of the L1 and L2 caches and some of the other storage structures are shared between PPE threads, but some storage structures are duplicated for each thread.

Both PPE threads share the execution units, microcode engine, instruction-fetch control, PPSS, plus the following caches, arrays, queues, and other storage structures:

- ERATs (Effective-address-to-Real-Address Translation arrays)—I-ERAT (instructions) and D-ERAT (data) and their associated ERAT Miss Queues.
- Translation Lookaside Buffer (TLB).
- L1 Caches (ICache and DCache).
- L2 Cache.
- Instruction prefetch request queue (IPFQ).
- Load-Miss Queue (LMQ).
- CIU Store Queue.
- NCU Store Queue.
- L2 Store Queue.

Each PPE thread has its own copy of the following arrays and queues, so that each thread can use that resource independently from the other thread:

- Segment Lookaside Buffer (SLB).
- Branch History Table (BHT), with global branch history.
- Instruction Buffer (IBuf).
- Link Stack.

6.1.1 Configuration

Of the PPE's storage structures, most can be configured by software to some extent using the following registers:

- Hardware Implementation Registers 1 and 4 (`HID1`, `HID4`).
- CIU Mode Setup Register (`CIU_ModeSetup`).
- NCU Mode Setup Register (`NCU_ModeSetup`).

However, software management of the following structures can be more extensive, as described later in this chapter:

- L1 Caches.
- L2 Cache, and its RC queues and RMT.
- ERATs.
- TLB and its RMT.
- Instruction Prefetch Queue.
- Load-Miss Queue.

All of the instructions for managing the L1 and L2 caches are accessible by problem-state (user) software, but RMTs (*Section 6.3* on page 154) are only accessible by privilege-2-state (supervisor) software, and ERATs (*Section 6.1.7* on page 150) are only accessible by privilege-1-state (hypervisor) software.

6.1.2 Overview of PPE Cache

The PPE's several storage structures are distributed throughout its PowerPC Processor Unit (PPU) and its PowerPC Processor Storage Subsystem (PPSS), as shown in *Figure 6-1* on page 135. This figure differentiates the structures according to their speed with respect to the full CBE Core Clock speed.

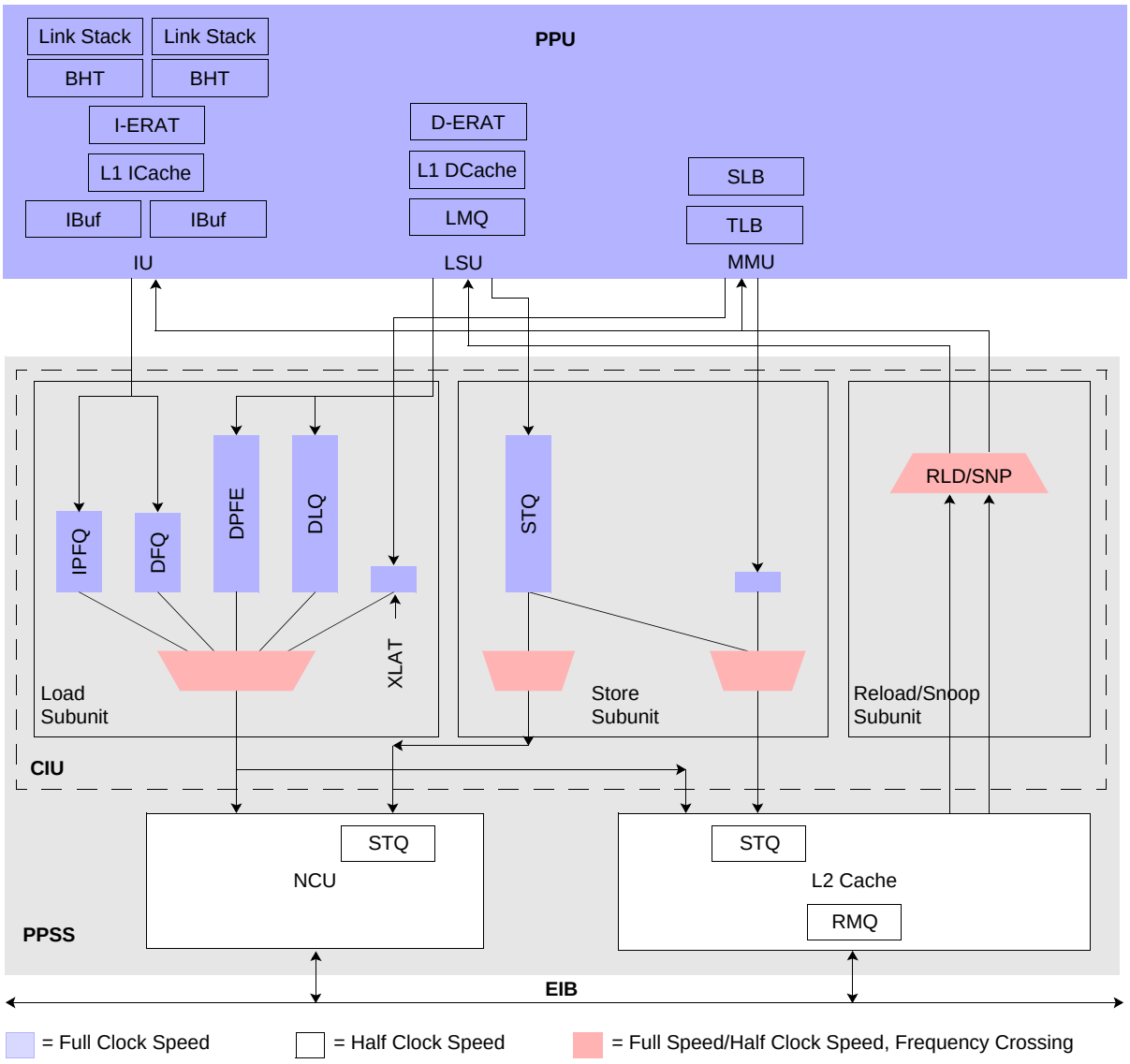
6.1.2.1 *PowerPC Processor Unit*

The major blocks of the PPU that support caches, arrays, queues, and other storage structures are the Instruction Unit (IU), Load and Store Unit (LSU), and Memory Management Unit (MMU):

- The IU contains the L1 ICache (32 KB), I-ERAT (64 entries), two BHTs (4096 x 2-bit entries and 6 bits of global branch history, per thread), two Link Stacks (4 entries, per thread), and two Instruction Buffers (IBufs, 4 instructions x 5 entries, per thread).
- The LSU contains the L1 DCache (32 KB), D-ERAT (64 entries), and Load-Miss Queue (LMQ, 8 entries).
- The MMU contains two SLBs (64 entries, per thread), and the TLB (1024 entries).

Section 4 Virtual Storage Environment on page 79 describes the workings of MMU caches. The PPU and all its subunits operate at the full frequency of the CBE Core Clock.

Figure 6-1. PPE Cache



= Full Clock Speed = Half Clock Speed = Full Speed/Half Clock Speed, Frequency Crossing

BHT	Branch History Table (with global history)
DFQ	Demand Instruction Request Queue
DLQ	Demand Data load Request Queue
DPFE	Data Prefetch Engine
EIB	Element Interconnect Bus
ERAT	Effective-to-Real Address Translation
IPFQ	Instruction Prefetch Request Queue
IU	Instruction Unit
L1 DCache	Level 1 Data Cache
L1 ICache	Level 1 Instruction Cache
L2	Level 2 Cache
LMQ	Load Miss Queue
LSU	Load and Store Unit
MMU	Memory Management Unit
NCU	NonCacheable Unit
PPSS	PowerPC Processor Storage Subsystem
PPU	PowerPC Processor Unit
RLD	Reload
RMQ	Reload Miss Queue
SLB	Segment Lookaside Buffer
SNP	Snoop
STQ	Store Queue
TLB	Translation Lookaside Buffer
XLAT	Translate Request Queue

6.1.2.2 *PowerPC Processor Storage Subsystem*

The major blocks of the PPSS, shown in *Figure 6-1*, are the Core Interface Unit (CIU), the Noncacheable Unit (NCU), and L2 cache.

The CIU contains Load, Store, and Reload Subunits, which—with the exception of a load-hit-store comparison—work independently of the each other. In the CIU:

- The Load Subunit contains the Instruction Prefetch Request Queue (IPFQ, 2 entries), Demand Instruction Request Queue (DFQ, 2 entries), Data Prefetch Engine (DPFE, 8 **dcbt** instructions), Demand Data Load Request Queue (DLQ, 8 entries), and MMU Translate Request Queue (XLAT, 1 entry).
- The Store Subunit contains a Store Queue (STQ, 8 x 16-byte entries).
- The Reload/Snoop Subunit contains a Reload/Snoop block (RLD/SNP) that performs 2:1 clock-frequency conversion and multiplexing of data from the L2 to the 32-byte reload bus that is shared by the LSU, IU, and MMU. A cache line is sent over four subsequent full-speed cycles totaling 128 bytes. Noncacheable data is sent in one cycle totaling 32 bytes. The Reload/Snoop subunit also performs 2:1 frequency conversion for the snoop bus.

The NCU has a Store Queue (STQ, 4 x 64-byte entries), and the L2 has a Store Queue (STQ, 8 x 64-byte entries). The L2 also has a Reload Miss Queue (RMQ, 6 x 128-byte cache-line entries). The CIU accesses the NCU or L2 based on the Caching-Inhibited (I) storage-attribute bit in the page table entry (PTE).

6.1.3 L1 Caches

The PPE has Level-1 (L1) instruction and data caches that support the *PowerPC Architecture* cache-control instructions. Cacheability of main-storage accesses (loads, stores, and instruction fetches) can be controlled by the caching-inhibited (I) bit in the page-table entry (PTE) for the accessed page. To be cacheable, the I bit must be cleared to ‘0’.

6.1.3.1 *Features*

- *L1 Instruction Cache (ICache):*
  - 32-KB, 2-way set-associative.
  - Read-only.
  - 128-byte cache-line size.
  - 5-entry, 4-instruction-wide instruction buffer, duplicated for each thread.
  - Out-of-order completion of cache-miss load instructions (otherwise in-order execution).
  - Effective-addressed index, real-addressed tags.
  - One read port, one write port (one read or write per cycle).
  - Does not support snoop operations.
  - Contents not guaranteed to be included in the L2 cache.
  - Cache bypass (inhibit) configuration options.
- *L1 Data (DCache):*
  - 32-KB, 4-way set-associative.

- Write-through.
- Not write-allocate.
- 128-byte cache-line size.
- 8-entry load-miss queue, shared between threads.
- 16-entry store queue, shared between threads.
- Nonblocking (hit under miss).
- Supports back-invalidation for snooping.
- Effective-addressed index, real-addressed tags.
- One read port, one write port (one read or write per cycle).
- Contents fully included in the L2 cache.
- Cache bypass (inhibit) configuration options.

Both L1 caches are dynamically shared by the two PPE threads; a cache block can be loaded by one thread and used by the other thread. The coherence block, like the cache-line size, is 128 bytes for all caches. The L1 ICache is an integral part of the PPE's Instruction Unit (IU), and the L1 DCache is an integral part of the PPE's Load/Store Unit (LSU), as shown in *Figure 2-2 PPE Functional Units* on page 46. Accesses by the processing units to the L1 caches occur at the full frequency of the CBE Core Clock.

The L1 caches support the *PowerPC Architecture* cache-management instructions. These perform operations such as write-back, invalidate, flush (write-back and invalidate), clear the contents to '0's, touch (which can be used by the compiler to speculatively load the cache line), and hint (that the cache line will probably be accessed soon). See *Section 6.1.6 Instructions for Managing the L1 and L2 Caches* on page 146 for a description. All are problem-state (user) instructions.

6.1.3.2 *ICache Instruction Prefetch*

The PPE uses speculative instruction prefetching (including branch targets) for the L1 ICache and instruction prefetching for the L2 cache. Because the PPE can fetch up to four instructions per cycle and can execute up to two instructions per cycle, the IU will often have several instructions queued in the instruction buffers. Letting the IU speculatively fetch ahead of execution means that L1 ICache misses can be detected early and fetched while the processor remains busy with instructions in the instruction buffers.

In the event of an L1 ICache miss, a request for the desired line is sent to the L2. In addition, the L1 ICache is also checked to see if it contains the next sequential cache line. If it does not, a prefetch request is made to the L2 to bring this next line into the L2 (but not into the L1, to avoid L1 ICache pollution). This prefetch occurs only if the original cache miss is committed (that is, all older instructions must have passed the execution-pipeline point—called the *flush point*—at which they can be written back to main storage). This is especially beneficial when a program jumps to a new or infrequently used section of code, or following a task switch, because prefetching the next sequential line into the L2 hides a portion of the main-storage access latency.

Prefetch requests are not made when a page boundary may potentially be crossed, meaning that the next sequential cache line is on a different aligned 4-KB boundary. Prefetching is also not performed on caching-inhibited instructions (I = ‘1’ in the PTE).

6.1.3.3 *ICache Fetch Misses*

When an L1 ICache miss occurs, a request is made to the L2 for the cache line. This is called a *demand fetch*. To improve performance, the first beat[11] of data returned by the L2 contains the fetch group with the address that was requested, and so returns the data *critical-sector first*. To reduce the latency of an L1 miss, this critical sector of the cache line is sent directly to the instruction pipeline, instead of first writing it into the L1 cache and then rereading it (this is termed *bypassing the cache*).

Each PPE thread can have up to one instruction demand-fetch and one instruction prefetch outstanding to the L2 at a time. This means that in multithread mode, up to four total instruction requests can be pending simultaneously. In multithreading mode, an L1 ICache miss for one thread does not disturb the other thread.

6.1.3.4 *ICache Replacement Algorithm*

The L1 ICache uses a true binary Least Recently Used (LRU) replacement policy, without Replacement Management Table (*Section 6.3* on page 154) locking.

6.1.3.5 *ICache Aliasing*

The ICache is 2-way set associative, so a line that is loaded can be put into one of two ways of a congruence class (set). The smallest page size for the 32-KB L1 ICache is 4 KB, so the same real address can map up to four different locations in the L1 ICache (two congruence classes, two ways in each class). This condition is referred to as *aliasing*. Because the ICache is read-only, no particular hazards exist as a result of this. However, the effectiveness of the L1 ICache may be lessened if a great deal of localized aliasing occurs, which is not common in most cases.

Aliasing in the L1 DCache are treated differently, as described in *Section 6.1.3.9* on page 139.

6.1.3.6 *ICache Invalidation*

To help invalidate the contents of the ICache efficiently, the PPE implements a special mode of operation for the **icbi** instruction. This mode can be selected with the Enable Forced **icbi** Match (`en_icbi`) bit in the Hardware Implementation Register 1 (`HID1`). In this mode, all directory lookups on behalf of an **icbi** act as though there were a real-address match, so that all cache lines looked at by the **icbi** are invalidated. As a result, the entire L1 ICache can be invalidated by issuing a series of **icbi** instructions that step through each congruence class of the ICache.

Another way to clear the ICache is to fill it with a set of known values by executing code that touches each cache line. One way to write this code is to have a series of 512 branches to branches whose EAs are sequentially separated by 128 bytes (the line size of the ICache).

The entire ICache can also be invalidated by first clearing the `HID1[flush_ic]` bit to ‘0’ and then setting the same bit to ‘1’.

11. A data beat is the data sent in one L2 clock cycle, which is two CBE Core-Clock cycles.

6.1.3.7 *DCache Load Misses*

Loads that miss the L1 DCache enter a load-miss queue for processing by the L2 cache and main storage. Data is returned from the L2 in 32-byte beats on four consecutive cycles. The first cycle contains the critical section of data, which is sent directly to the register file. The DCache is occupied for two consecutive cycles while the reload is written to the DCache, half a line at a time. The DCache tag array is then updated on the next cycle, and all instruction issue is stalled for these three cycles. In addition, no instructions can be recycled during this time. The load-miss queue entry can then be used seven cycles after the last beat of data returns from the L2 to handle another request.

Instructions that are dependent on a load are issued speculatively, assuming a load hit. If it is later determined that the load missed the L1 DCache, younger instructions and any instructions that are dependent on the load are flushed, refetched, and held at dispatch until the load data has been returned. This behavior allows the PPE to send multiple overlapping loads to the L2 without stalling if they are dependent. In multithreading mode, this behavior allows load misses from one thread to occur without stalling the other thread.

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

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

6.1.3.8 *DCache Replacement Algorithm*

The L1 DCache uses a pseudo Least Recently Used (p-LRU) replacement policy. It is a 4-way policy, comparable to the L2 pseudo-LRU 8-way policy described in *Section 6.1.5.3* on page 143, but without Replacement Management Table (*Section 6.3* on page 154) locking. The policy is updated whenever a line is read in the cache. The L1 DCache never needs to cast out data because it uses a write-through scheme (stores are sent to the L2 as well as the L1 DCache).

6.1.3.9 *DCache Indexing*

The DCache is indexed with effective addresses. The data array and cache directory both have a single read and write port (one read or write per cycle). The data and tag arrays operate independently, giving the cache some of the features of having dual ports. Specifically, the cache allows a store to check the directory tags while a previous store writes into the data array.

The DCache tags and the D-ERAT are checked in parallel, similar to the way the ICache is accessed.

6.1.3.10 *DCache Aliasing*

DCache aliasing occurs when separate translation entries have the same real address (RA) mapped from different effective addresses (EAs). Data can be in any of four ways at each of two congruence classes. If the data is not at the congruence class determined by EA[51], but there is

a hit in the cache, the data is found at the other congruence class, addressed by the NOT EA[51]. That line is invalidated and the data is brought in from the L2 and placed at the congruence class determined by EA[51].

If shared memory is used for communication between simultaneously running threads or two processes on the same PPE thread, mapping the shared memory area to the same EA in both threads will prevent unnecessary cache invalidations, as described above.

6.1.3.11 *DCache Write-Back and Invalidation*

The L1 DCache is a write-through design, so that it never holds data in a modified state. As a result, to perform a write-back of the L1 DCache, the only instruction required is a **sync**, which will force any pending stores in the store queue above the L1 cache to become globally coherent before the **sync** is allowed to complete.

A sticky mode bit is provided in the Hardware Implementation Register 4 (`HID4`) to invalidate the entire contents of the L1 DCache. To invalidate the cache, software must clear the `HID4[l1dc_flsh]` bit to '0', then set the same bit to '1', then execute a **sync** instruction.

6.1.3.12 *DCache Boundary-Crossing*

The L1 DCache is physically implemented with 32-byte sectors. Thus, conditionally microcoded load or store instructions (*Table A-1 PowerPC Instructions, by Execution Unit* on page 712) that attempt to perform a L1 DCache access that crosses a 32-byte boundary must be split into several instructions. When one of these misaligned loads or stores first attempts to access the L1 DCache, the misalignment is detected and the pipeline is flushed. The flushed load or store is then refetched, converted to microcode at the decode stage, and split into the appropriate loads or stores, as well as any instructions needed to merge the values together into a single register.

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

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

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

The load string instruction first attempt to use load word instructions to move the data. If the access would cross a 32-byte boundary when it accesses the L1 DCache, the load is flushed and refetched, and it proceeds byte-by-byte. The store string instruction behaves similarly.

6.1.3.13 *L1 Bypass (Inhibit) Configuration*

To bypass the L1 and L2 caches (that is, to inhibit their use), set the `HID4[enb_force_ci]` bit to '1' and the `HID4[dis_force_ci]` bit to '0'.

6.1.4 Branch History Table and Link Stack

The PPE's Instruction Unit (IU) is shown in *Figure 6-1* on page 135. For each thread, the IU has a Branch History Table (BHT) with 4096 x 2-bit entries and 6 bits of global branch history, and a 4-entry link stack. The link stack is used to predict the target address of branch-to-link instructions. The following mode bits enable these functions:

- Branch History Table: Hardware Implementation Register 1, `HID1[bht_pm]` bit.
- Global Branch History: Hardware Implementation Register 1, `HID1[dis_gshare]` bit.
- Link Stack: Hardware Implementation Register 1, `HID1[en_ls]` bit.

Static branch prediction can reduce pollution in the BHT, improving accuracy on branches elsewhere in the code. The BHT is accessed in parallel with the L1 ICache. The global branch history is constantly updated with the latest speculatively-predicted history value. In the event of a pipeline flush including branch misprediction, the correct history state is restored. Using two bits for each BHT entry allows the processor to store weakly and strongly taken or not taken states.

Only instructions that relied on the BHT to predict their direction will ever cause an update to the BHT, meaning that unconditional branches (including conditional branches with BO = 1z1zz), and statically predicted conditional branches do not update the BHT.

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

6.1.5 L2 Cache

The PPE contains a 512-KB unified Level-2 (L2) cache. The L2 supports the *PowerPC Architecture* cache-control instructions and it controls cacheability with the caching-inhibited (I) bit in PTEs.

6.1.5.1 *Features*

- 512-KB, 8-way set-associative or direct-mapped[12].
- Unified instruction and data caching.
- Inclusive of L1 DCache. Not inclusive of L1 ICache.
- Write-back (copy-back).
- 128-byte cache-line size.
- 128-byte coherence granularity.
- Allocation on store miss.

12. See *Section 6.1.5.2* on page 142 for configuring the L2 cache to operate in direct-mapped mode.

- Choice of replacement algorithms: True binary LRU, pseudo-LRU with Replacement Management Table (RMT) locking[13], or direct-mapped.
- Hardware coherency (MERSI plus Mu and T), with separate directory for snoops.
- Critical quadword-forwarding on data loads and instruction fetches.
- 6-entry, 128-byte reload/store-miss queue (Read-and-Claim queues).
- 6-entry, 128-byte castout queue.
- 8-entry, 64-byte fully associative store queue.
- 4-entry, 128-byte snoop intervention/push queue.
- Store-gathering.
- Nonblocking L1 DCache invalidations.
- Real-address index and tags.
- One read port, one write port (one read or write per cycle).
- Separate snoop directory for all system bus snoops.
- Bypass (inhibit) configuration option.
- Error correction code (ECC) on data.
- Parity on directory tags (recoverable using redundant directories).
- Global and dynamic power management.

The L2 maintains full cache-line coherency within the system and can supply data to other processor elements. Logically, the L2 is an inline cache. It is a write-back cache that includes all of the L1 DCache contents but is not guaranteed to include the L1 ICache contents. The L2 is dynamically shared by the two PPE execution threads; a cache block can be loaded by one thread and used by the other thread.

The L2 cache handles all cacheable loads and stores (including **lwarx**, **ldarx**, **stwcx.**, and **stdcx.** instructions), data prefetches, instruction fetches, instruction prefetches, cache operations, and barrier operations. The L2 cache supports the same cache-management instructions as are supported by the L1 caches, as described in *Section 6.1.6 Instructions for Managing the L1 and L2 Caches* on page 146.

6.1.5.2 *Cache Configuration*

The basic L2 cache operation can be configured in the following registers:

- As an alternative to the 8-way set-associative configuration, the L2 cache can be configured to operate in direct-mapped mode by setting the `L2_ModeSetup1[RMT_Mode]` register field to ‘11’.
- L2 cache replacement algorithm can be configured in the `L2_ModeSetup1[RMT_Mode]` register field. See *Section 6.1.5.3 Replacement Algorithm* on page 143.
- L2 cache operation during errors can be configured in the `L2_ModeSetup1[L2_Stop_Data_Derr_En]`, `L2_ModeSetup1[L2_Stop_Ins_Derr_Dis]`, and `L2_ModeSetup1[L2_Stop_TO_Dis]` register bits.

13. See *Section 6.3.2* on page 157 for a description of RMT.

- The L1 and L2 caches can be disabled using the `HID4[dis_force_ci]` and `HID4[enb_force_ci]` register bits.

6.1.5.3 *Replacement Algorithm*

One of three L2 cache-replacement algorithms can be selected by the `L2_ModeSetup1[RMT_Mode]` field:

- *Binary Least-Recently Used (LRU) Mode*—This algorithm is based on the binary tree scheme.
- *Pseudo LRU (p-LRU) Mode*—This algorithm is described immediately below, in this section.
- *Direct-Mapped Mode*—This algorithm uses three tag address bits to map an address to one of eight congruence-class members.

The Pseudo LRU (p-LRU) mode uses software-configureable Address Range Registers and a Replacement Management Table (*Section 6.3.1.3 Address Range Registers* on page 155 and *Section 6.3 Replacement Management Tables* on page 154) to lock cache ways to a particular Replacement Class Identifier (`RclassID`). The other two modes cannot use these facilities.

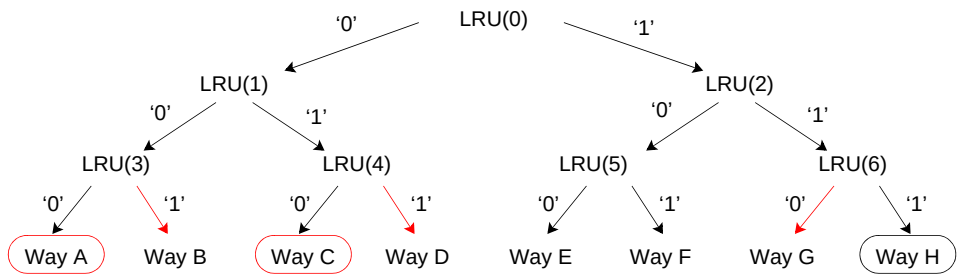
The p-LRU mode supports the following features:

- *Multiple Cache-Line Locking*—The L2 LRU replacement requirement is used to perform cache-line replacement with the normal LRU method and with cache ways locked by the RMT.
- *RclassID from PPU to LRU Controller*—The LRU controller uses the `RclassID` from the PPU to determine which way or ways are locked by the RMT.
- *Cache-Line Replacement Involves RMT in the LRU Controller*—The LRU controller must look up the LRU value for the corresponding cache line and then update the LRU value based on the look-up value from the RMT, as described in *Section 6.3.2 PPE L2 Replacement Management Table* on page 157.
- *Updating LRU Data (Pointer) for Unlocked Cache Lines*—The LRU controller also must update the least-recently-used pointer to the next cache line that is unlocked by the RMT.

Figure 6-2 shows an example of Pseudo LRU replacement. In this figure, if the way replacement value in the RMT is ‘0101 1110’, then:

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

Figure 6-2. Pseudo-LRU Binary Tree for L2-Cache LRU Binary Tree with Locked Ways A, C, and H



6.1.5.4 *Cache Writeback*

The following procedure writes back the entire L2 cache to main storage by means of software. The other thread should be quiesced or guaranteed not to be performing cacheable operations.

1. Set the `L2_ModeSetup1` register mode bits to direct-mapped mode (`L2_ModeSetup1[RMT_Mode]` = '11'). Switching between LRU mode and direct-mapped mode can be done at any time. Data integrity is maintained, although performance might be affected.
2. Pick a 4-MB region of main storage to hit all eight ways in direct-mapped mode.
3. Execute one load, store, or **dcbz** instruction to byte(s) in each 128-byte cache line in the chosen main-storage region. This fills the L2 with the contents of the chosen region and forces out all other data contained in the cache. However, if the address range of the current contents of the L2 cache is unknown, and the L2 cache needs to be written back to memory as is, then use only a load instruction (not a store or **dcbz**).
4. Execute one **dcbf** instruction for each 128-byte cache line in the specified main-storage region. This writes back and invalidates the specified region. If the contents of the region were resident and modified in the L2, the **dcbf** operation ensures that the data is copied back to main storage. Instruction fetches can cause the L2 cache to retain some valid data even after the **dcbf** operation is performed.

6.1.5.5 *Cache Queues*

The 512-KB L2 cache is physically implemented as four quadrants, each quadrant containing 32 bytes of the 128-byte cache line. The cache contains the following queues:

- CIU Load Queue
  - I-Demand Queue (2 entries).
  - I-Prefetch Queue (2 entries).
  - Translate Queue (1 entry).
  - D-Demand Queue (8 entries).
  - D-Prefetch Queue (8 entries).
- CIU Store Queue: 8 x 16 bytes.
- L2 Store Queue: 8 x 64 bytes, gather enable.

- L2 RC Reload Queue: 6 x 128 bytes.
- L2 Snoop Queue: 4 x 128 bytes.
- L2 Castout Queue: 6 x 128 bytes.

A cache-line read from the L2 requires two data beats (two L2 clock cycles, which is four CBE Core-Clock cycles) and returns 16 bytes of data per beat from each quadrant. The L2 then sends two 64-byte beats to the CIU data-reload bus. A cache-line read request that misses the L2 is sent to the EIB. Reload data coming from the EIB is received in eight back-to-back 16-byte data beats. Because all reload data is sent back-to-back, the L2 must wait for all missed data from the EIB before sending data to the PPU. The L2 allows the PPU to request which quadword it wants to receive first from the L2 (called the *critical quadword*). A cache-line write requires two write accesses to the L2 cache, performing a 64-byte write for each access.

6.1.5.6 *Read-and-Claim State Machines*

The L2 cache has six Read-and-Claim (RC) state machines, with corresponding address and data queues. They manipulate data in and out of the L2 cache in response to PPU and/or snoop requests. These queues have the following features:

- *Configuration*—The number of RC state machines used for prefetches can be configured in the Prefetch Outstanding Request Limit (`PR`) field of the CIU Mode Setup Register (`CIU_ModeSetup`). When the RC state-machine resource that is servicing data or instruction prefetches reaches a threshold, the data and instruction prefetches are temporarily suspended until the RC state-machine resources become available again. This threshold and other parameters of prefetching can be set in the `CIU_ModeSetup` register. See *Section 6.1.9* on page 150 for more information.
- *Congruence Classes*—Two RC state machines never simultaneously work on the same L2 congruence class. This prevents LRU victim conflicts where both RC state machines have chosen the same victim.
- *Castouts*—If an RC state machine starts on a miss and the victim is valid, the Castout (CO) state machine must be available for the RC state machine to process the access. If the CO state machine is busy working on a prior castout, the RC state machine does not acknowledge the access if it is a miss (it will be reissued by the CIU). If it is a miss and the CO state machine is available to process the castout, the RC state machine accepts the access (acknowledge the CIU Request). After the RC state machine has finished, it can process a new access that results in a hit, even though the CO state machine is still processing a castout from a prior access.

6.1.5.7 *Store Queue*

The L2 store queue consists of eight fully associative 64-byte sectors. Each sector is byte writable, so gathering occurs at the point the buffer is written. This is effectively a store cache with the following features:

- *Store Gathering Logic*—Gathering is supported for nonatomic, cacheable stores. Gathering of nonstore opcodes is not supported (for example, a request that involves a **dcbz**, PTE update, or atomic stores is not gathered). Gathering is done into any of the store queue entries, to any 64-byte combination within an entry, for any store operation, assuming no barrier operations prevent gathering to a previously stored entry.

- *Store Queue*—A request from the CIU store queue is sent in two consecutive cycles (address, then data) to the L2 and is stored in one of the eight 64-byte store queue entries. Unloading of the eight entries in the store queue is done in a FIFO manner.
- *Store Queue Request by Read-and-Claim (RC) State Machine*—No two RC state machines (*Section 6.1.5.6*) are simultaneously active to the same L2 congruence class. Therefore, the store queue sometimes does not dispatch until a previous store is completed by an RC state machine, even if another RC state machine is available.

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

Store-queue data parity and other errors can be read in the L2 Cache Fault Isolation Register (`L2_FIR`).

6.1.5.8 *Reservations*

The L2 cache supports two reservation bits (one per thread) to handle a load and reserve request by each thread. The snoop logic compares against both reservation addresses, but this is not a thread-based comparison.

6.1.5.9 *L2 Bypass (Inhibit) Configuration*

To bypass the L1 and L2 caches (that is, to inhibit their use), set the `HID4[enb_force_ci]` bit to ‘1’ and clear the `HID1[en_if_cach]` bit to ‘0’.

6.1.6 Instructions for Managing the L1 and L2 Caches

The PPE supports six *PowerPC Architecture* instructions for management of the L1 and L2 caches. All are problem-state (user) instructions. These are described in the sections below.

The *PowerPC Architecture* provides cache management instructions that allow user programs to:

- Invalidate an ICache line (**icbi**).
- Hint that a program will probably access a specified DCache line soon (**dcbt**, **dcbtst**).
- Set the contents of a DCache line to ‘0’s (**dcbz**).
- Copy a modified DCache line to main storage (**dcbst**).
- Copy a modified DCache line to main storage and invalidate the copy in the DCache (**dcbf**).

The cache-management instructions are useful in optimizing the use of main-storage bandwidth in such applications as graphics and numerically intensive computing. The functions performed by these instructions depend on the storage-control attributes associated with the specified storage location (see *Section 4.2.6.7 WIMG-Bit Storage Control Attributes* on page 91). Cache management is also needed when a program generates or modifies code that will be executed (that is, when the program modifies data in storage and then attempts to execute the modified data as instructions).

6.1.6.1 *Instruction Cache Block Invalidate*

The L1 ICache line size for the **icbi** instruction is 128 bytes. The L1 ICache does not support a snoop operation. Because of this, the L1 ICache is not necessarily consistent with modifications to those storage locations.

The effective address of the **icbi** instruction is translated as a data address by the Load Store Unit (LSU). Because of this, data-address translation rules are used, and any exception that results is handled as a Data Storage exception. The address translation is done using the D-ERAT, not the I-ERAT (see *Section 4.2.4 Effective-to-Real-Address Translation* on page 83). Because this instruction invalidates cache lines based on real addresses, it must search and invalidate up to eight cache lines in the L1 ICache to account for the potential aliasing conditions described in *Section 6.1.3.4* on page 138. Two passes are done to invalidate the entries, one to generate a vector containing the lines that will be invalidated, and another pass to do the actual invalidation.

In addition to checking and invalidating lines L1 ICache lines, the **icbi** instruction and the real address of the line to be invalidated are broadcast to all PPEs in the system. This PPE must also accept **icbi** instructions from other PPE processors.

6.1.6.2 *Data Cache Block Touch*

The DCache line size for the **dcbt** and **dcbtst** instructions is 128 bytes. The behavior of **dcbtst** is the same as the behavior of **dcbt** with the Touch Hint (TH) field equal to ‘0000’, as described below.

These instructions act like a load instruction from the viewpoint of the cache hierarchy and the TLB, with the following exceptions:

- If data translation is enabled (`MSR[DR]` = ‘1’), and a segment or page fault results, the instruction is cancelled and no exception occurs.
- If a TLB miss results and `LPCR[TL]` = ‘0’ (hardware table-lookup mode), the TLB is reloaded and the corresponding reference bit is set if a matching PTE is found.
- If the page-protection attributes prohibit access, the page is marked caching-inhibited (I = ‘1’), `LPCR[TL]` = ‘1’ (software table-lookup mode), or the page is marked guarded (G = ‘1’), the instruction is cancelled and does not reload the cache.

Otherwise, the instruction checks the state of the L1 DCache, and if the data requested is not present, a reload request is sent to the L2. If the data is present in the L2, the data is forwarded to the L1. If the data is not present, the L2 requests it from main storage and only the L2 is reloaded (the data is not forwarded to the L1), except for the case in which TH = ‘0000’. If the **dcbt** or **dcbtst** instruction reloads a cache line, the replacement attributes for the cache (for example, the least-recently used information) are updated.

The PPE implements the optional form of the **dcbt** instruction defined in the *PowerPC Virtual Environment Architecture, Book II*. This form of the instruction can set up a *data stream*. Each data stream is managed independently. There are a total of eight streams available. These are split into four streams for each thread. The Touch Hint (TH) field of the instruction has the following implementation-specific characteristics:

- The PPE implements the TH = ‘0000’, ‘1000’, and ‘1010’ forms.
- All other TH(0) = ‘0’ forms are not supported and are implemented the same as TH = ‘0000’.

- All other TH(0) = ‘1’ forms are not supported and are ignored.
- For the TH = ‘1010’ form, the Transient field (bit 57 of the effective address) is ignored.

The PPE’s data-prefetch engine (DPFE) implements the **dcbt** and **dcbst** instructions. The prefetch engine services all active streams simultaneously in a round-robin fashion. Each time a stream is serviced, a prefetch request is generated for the stream and issued to the L2 cache. One cache line is requested at a time. By default, data prefetch has the lowest priority of any queue in the PPE’s Load subunit. This default behavior can be changed. See *Section 6.1.10 Load-Subunit Management* on page 150.

The basic function of the **dcbt** and **dcbtst** instructions can be changed in the following ways:

- Force all load instructions to be treated as loading to guarded storage (thus causing data cache touch instructions, such as **dcbt**, to be treated as nop instructions) using the `HID4[force_geq1]` register bit.
- Force the x-form of the **dcbt** and **dcbtst** instructions to be treated as nop instructions using the `HID4[tch_nop]` register bit.

6.1.6.3 *Data Streams*

As described in *Section 6.1.6.2*, the PPE implements the optional *data stream* form of the **dcbt** and **dcbtst** instructions. To improve prefetch efficiency, the L2 notifies the Core Interface Unit (CIU), *Section 6.1.2.2* on page 136, when it detects that a demand load to the same prefetch address is being serviced by the L2. In this case, the demand load is ahead of the data prefetch and the prefetch is terminated. The CIU does not terminate the stream, but it can then send another load request.

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

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

6.1.6.4 *Data Cache Block Zero*

In the CBE implementation, the DCache line size for the **dcbz** instruction is 128 bytes. The function of **dcbz** is performed in the L2 cache. If the line addressed by the **dcbz** is present in the L1 DCache, the line is invalidated before the operation is sent to the L2 cache. The L2 cache gains exclusive access to the line, without actually reading the old data, and performs the zeroing function in a broadside manner. That is, all bits in the cache line are cleared to ‘0’, which means the data cannot be discovered by another process.The **dcbz** instruction is handled like a store and is performed in the appropriate storage-model order. It does not bypass any of the store queues. In general, the performance of **dcbz** is better than a sequence of 8-byte stores, because it zeros 128-bytes at a time.

Regardless of whether the cache line specified by the **dcbz** instruction contains an error (even one that is not correctable with ECC), the contents of the line are cleared to ‘0’ in the L2 cache. If the line addressed by the **dcbz** instruction is in a memory region marked caching-inhibited (I = ‘1’), or if the L1 DCache is disabled (`HID4[en_dcway]` = ‘0000’), the instruction causes an Alignment interrupt.

6.1.6.5 *Data Cache Block Flush and Data Cache Block Store*

The L1 DCache line size for the **dcbf** and **dcbst** instructions is 128 bytes. A **dcbf** instruction causes all caches for all processor elements in the system to write modified contents back to main storage and invalidate the data in the cache for the line containing the byte addressed by the instruction. A **dcbst** behaves the same, except that the data in the cache remains valid (if present).

The **dcbf** and **dcbst** instructions will probably complete before the operation they cause has completed. A context-synchronizing instruction (see *Section 20 Shared-Storage Synchronization* on page 551) ensures that the effects of these instructions are complete for the processor element issuing the synchronizing instruction.

6.1.6.6 *Vector/SIMD Multimedia Extension Intrinsics*

The following instructions defined in the *AltiVec Technology Programming Interface Manual* are deprecated and are implemented as nops on the CBE processor:

- vec_dss(a)—Vector Data Stream Stop.
- vec_dssall()—Vector Stream Stop All.
- vec_dst(a,b,c)—Vector Data Stream Touch.
- vec_dstst(a,b,c)—Vector Data Stream Touch for Store.
- vec_dststt(a,b,c)—Vector Data Stream Touch for Store Transient.
- vec_dstt(a,b,c)—Vector Data Stream Touch Transient.

Software should use the PPE the **dcbt** and **dcbtst** instructions for stream control.

6.1.7 Effective-to-Real-Address Translation Arrays

The PPE contains two 64-entry, 2-way set-associative, Effective-to-Real-Address Translation (ERAT) arrays: an Instruction ERAT (I-ERAT) and a Data ERAT (D-ERAT). Both are shared by both threads. The ERATs store recently used EA-to-RA translations, and ERAT operation is maintained by hardware, as described in *Section 4.2.4 Effective-to-Real-Address Translation* on page 83.

6.1.8 Translation Lookaside Buffer

The PPE's Memory Management Unit (MMU) contains a1024-entry, 4-way set associative TLB cache that holds most-recently used page table entries (PTEs). There are 256 congruence classes (sets), each requiring eight bits for the TLB index. Hardware or software can be used to replace the TLB entry. Privileged PPE software can maintain the TLB based on searches of the PPE's page table, using access to MMIO registers in the MMU. For details, see *Section 4.2.7 Translation Lookaside Buffer* on page 92.

6.1.9 Instruction-Prefetch Queue Management

The Instruction Unit (IU) hardware examines each committed instruction fetch and checks to see if the next instruction is in the L1 ICache. If not, it issues a prefetch request.

When the L2 cache is available for a request, and no higher-priority request exists (see *Section 6.1.10* on page 150), the instruction prefetch queue forwards the prefetch request to the L2. The L2 loads the line from main storage, if necessary. There is no subsequent reload for this transaction, and data is not loaded into the L1.

When the L2 Read-and-Claim (RC) state machine (*Section 6.1.5.6* on page 145) resource that is servicing data prefetches or instruction prefetches reaches a threshold, the data and instruction prefetch are temporarily suspended until the RC state machine resources become available again. This threshold and other parameters of prefetching can be set in the `CIU_ModeSetup` register.

6.1.10 Load-Subunit Management

The Load Subunit in the CIU (*Figure 6-1* on page 135) employs a two-level arbitration scheme. The first level of arbitration is within each separate subqueue of the Load subunit. This arbitration is always from oldest entry to youngest entry. The second level of arbitration enforces a priority among queues. The default priority order is:

- Translate request queue (XLAT).
- Demand data load request queue (DLQ).
- Demand instruction request queue (DFQ).
- Instruction prefetch request queue (IPFQ).
- Data prefetch engine (DPFE).

This second-level priority scheme can be programmed in the `CIU_ModeSetup` register as follows (both settings are independent of the other):

- The priority of DLQ and DFQ can be swapped to change the order to: XLAT, DFQ, DLQ, IPFQ, DPFE.
- To prevent starvation, the DPFE priority can be raised to the second priority: XLAT, DPFE, DLQ, DFQ, IPFQ. Alternatively, the DPFE priority can be periodically toggled from second to fifth priority after 32 load requests are issued to the L2.

6.2 SPE Caches

Each SPE's MFC has the following caches, both of which can be managed by privileged PPE software:

- Translation Lookaside Buffer (TLB).
- Atomic (ATO) Unit Cache, called the *Atomic cache*.

6.2.1 Translation Lookaside Buffer

Each SPE's Synergistic Memory Management (SMM) unit, in the MFC, contains an 256-entry, 4-way, set-associative TLB cache that holds most-recently used PTEs. There are 64 congruence classes (sets), each requiring six bits for the TLB index. The TLB array is parity protected. Parity generation and checking can be disabled in the `SMM_HID` register. Hardware or software can be used to replace the TLB entry. For details, see *Section 4.3.5 Translation Lookaside Buffer* on page 107.

6.2.2 Atomic Unit and Cache

Each SPE's MFC contains an Atomic Unit that handles semaphore operations for the SPU and provides PTE data to the SMM. More specifically, the Atomic Unit:

- Provides atomic operations for the ***getllar***, ***putllc***, ***putlluc***, and **putqlluc** MFC atomic commands.
- Provides PTEs to the SMM for hardware table lookups and updates to the Reference (R) or Change (C) bits in PTEs (called an *RC update*).
- Maintains cache coherency by supporting snoop operations.

The Atomic cache stores six 128-byte cache lines of data. Four of those cache lines support semaphore operations, one cache line supports PTE accesses, and one cache line supports reloading data from cache-miss loads like **getllar**.

MFC atomic commands are issued from the DMA Controller (DMAC) to the Atomic Unit. These requests are executed one at a time. The Atomic Unit can have up to two outstanding requests: one immediate-form command (***getllar***, ***putllc***, or ***putlluc***) and one queued-form command (**putqlluc**). The SMM's PTE request can occur at the same time as MFC atomic requests. However, the Atomic Unit handles this situation by alternating SMM requests and MFC atomic requests.

6.2.2.1 *Atomic Unit Operations*

The Atomic Unit accepts MFC atomic requests, SMM requests, and snoop requests. The MFC atomic requests are ***getllar***, ***putllc***, and ***putlluc***. The SMM request is for a PPE page-table lookup with RC update. These requests are executed in the Atomic Unit RC machine. Each of these requests can come at any time. Two MFC atomic requests can be queued in the Atomic Unit while the first request executes. Only one SMM request can be pending at a time. Snoop requests can come every two cycles, at most.

The Get Lock Line and Reserve command, **getllar**, is similar to the *PowerPC Architecture* **lwarx** instruction. A 128-byte cache line of atomic data is loaded into the LS and sets a reservation in the Atomic Unit. Only one reservation is allowed at a time in the Atomic Unit. In the case of a **getllar** miss with the Atomic cache full, room is created for this load by casting out LRU data. Looping on the **getllar** MFC atomic command is not necessary, because software can look for the reservation lost event in the `MFC_RdAtomicStat` channel's lock-line command status.

The Put Lock Line Conditional command, **putllc**, is similar to the *PowerPC Architecture* **stwcx** instruction. This is a store command with a condition on reservation active. If the **putllc** data is a cache hit, the Atomic Unit executes the store by sending store data to the Atomic cache from the LS. In addition, it resets the reservation and sends success status to the DMAC, which, in turn, is routed to the SPU.

The Put Lock-Line Unconditional command, **putlluc**, is a store command without any condition on the reservation. The Atomic Unit executes this request as it does **putllc**, except that it does not send any status to the DMAC. Also, the Atomic Unit executes the store, regardless of the reservation flag.

The Put Queued Lock-Line Unconditional command, **putqlluc**, is issued to the Atomic Unit as a **putlluc** MFC atomic request. The Atomic Unit can have up to two outstanding requests: one immediate form command (***getllar***, ***putllc***, or ***putlluc***) and one queued form command (**putqlluc**).

6.2.2.2 *Lock Acquisition Sequence and its Cache Effects*

Significant performance improvement is realized by using the **putlluc** command to release locks. Using the **putlluc** command rather than a DMA **put** command results in the modification of the data in the Atomic cache. There is no need to invalidate the cache and update main storage if the lock-line block is still in the cache.

Because the SPE does not implement an SL1 cache, the **putlluc** command results in a direct store to the referenced real memory and the invalidation of all other caches that contain the affected line. The **getllar**, **putllc**, and **putlluc** commands support high-performance lock acquisition and release between SPEs and the PPE by performing direct cache-to-cache transfers that never leave the CBE processor.

The possible Atomic-cache state transitions based on a typical lock code sequence are:

1. Issue **getllar** command.
2. Read the `MFC_RdAtomicStat` channel to see if **getllar** done.
3. Compare (check for active lock).
4. Branch if not equal (If the lock is still set, go to the step 1).

5. Issue **putllc** (attempt to set lock).
6. Check to see if the ***putllc*** is done.
7. Branch if not equal (If the **put** conditional is not successful, go to step 1).
8. Critical code.
9. Issue ***putlluc*** (release lock).
10. Check if ***putlluc*** is done (for future Atomic command issue).

6.2.2.3 *Reservation Lost Events and their Cache Effects*

A reservation can be lost (cleared) in any of the following situations; some of these situations affect the state of the Atomic cache:

- One ***getllar*** request (***getllar*** A) of Atomic Unit 1 sets the reservation. Before the first ***putllc*** request (***putllc*** A) is issued by Atomic Unit 1, another ***getllar*** request (***getllar*** B) is received in Atomic Unit 1 to clear the reservation of ***getllar*** A and set a new reservation for a second ***getllar*** request (***getllar*** B).
- A ***putllc*** request is received for a different storage location for which the previous ***getllar*** established a reservation. This scenario occurs only if ***getllar*** A is executed, a reservation is set, and software is interrupted inside the interrupt routine. A ***putllc*** request to a different storage location resets the reservation, but the store is not performed. If there is no interrupt, a scenario in which a ***getllar*** and ***putllc*** pair does not address the same storage location would be a software error.
- SPU1 issued ***getllar*** command A and got a reservation. Processor Element 2 wants to perform a store, and the SPU1 ***putllc*** command has not been issued.
- SPU1 issued ***getllar*** command A and got the reservation. Processor Element 2 wants to perform a store that hits a Shared line in its Processor Element 2's cache. The SPU1 copy is invalidated before the ***putllc*** instruction is issued and performed.

See the *PowerPC Virtual Environment Architecture, Book II* for more details on reservations.

6.2.2.4 *Atomic-Cache Commands*

The *Cell Broadband Engine Architecture* defines an SL1 (SPU Level 1) cache for DMA transfers between LS and main storage. The architecture also defines five MFC commands for the SL1 cache: **sdcrt**, **sdcrtst**, **sdcrz**, **sdcrst**, and **sdcrf**. However, the CBE processor does not implement an SL1 cache.The **sdcrt** and **sdcrtst** commands are implemented in the SPE as nops, but three of the commands —**sdcrz**, **sdcrst**, and **sdcrf**—affect the SPE Atomic cache, as shown in *Table 6-1*:

Table 6-1. Atomic-Unit Commands

CBEA SL1 Command	SPE Atomic-Unit Command	Definition
sdcrf	**sdcrf**	Atomic-cache range write-back and invalidate.
sdcrt	nop	Undefined.

Table 6-1. Atomic-Unit Commands

CBEA SL1 Command	SPE Atomic-Unit Command	Definition
sdcrtst	nop	Undefined.
sdcrst	**sdcrst**	Atomic-cache range store.
sdcrz	**sdcrz**	Atomic-cache range cleared to ‘0’.

6.2.2.5 *Atomic-Cache Flush*

Setting the Flush (`F`) bit of the `MFC_Atomic_Flush` register activates an Atomic-cache flush that copies cache lines in the Modified state back to main storage. The RC machine executes an Atomic-cache flush by dispatching the castout machine six times to copy all six entries of the Atomic cache back to main storage.

While the Atomic Unit performs a cache flush, the SPU must suspend the DMAC so that no DMAC atomic or SMM requests can enter the Atomic Unit. Otherwise, the Atomic Unit flags a hardware error called *ATO Flush Collision*, reported in the MFC Fault Isolation (`MFC_FIR`) register. See the *Cell Broadband Engine Registers* for details.

6.3 Replacement Management Tables

The PPE and SPEs provide a method for supervisor-state software to control L2 and TLB cache replacements, based on a Replacement Class ID (`RclassID`). This class ID is used as an index into a Replacement Management Table (RMT), which is used to lock entries in the L2 and TLB caches. The locking function of the RMTs modify the way in which the L2 and TLB pseudo-LRU algorithms operate.

The PPE has an RMT for managing its TLB, described in *Section 6.3.1* on page 154, and for its L2 cache, described in *Section 6.3.2* on page 157. Each SPE also has an RMT for managing its TLB, described in *Section 6.3.3* on page 158.

The RMTs are useful when a small set of pages are frequently accessed by application software and need to be locked in the L2 and/or TLB to prevent misses—for example, in real-time applications (see *Section 11.2.3.2* on page 336 for more detail). They are also useful for preventing streaming data from casting out other data. However, overlocking resources can negatively impact performance.

6.3.1 PPE TLB Replacement Management Table

The PPE’s TLB RMT is stored in the PPE Translation Lookaside Buffer RMT Register (`PPE_TLB_RMT`). This RMT can lock translation entries into the TLB by reserving particular ways of the TLB for specific EA ranges in a program. TLB replacement management applies only to the hardware-managed TLB mode, as described in *Section 4.2.7.1 Enabling Hardware or Software TLB Management* on page 92 and *Section 4.2.7.3 TLB Replacement Policy* on page 94. In the software-managed TLB mode, software controls TLB-entry replacement.

6.3.1.1 *RClassID Generation*

In the PPE, a 3-bit Replacement Class ID (`RclassID`) is generated from the effective address specified by instruction fetches or by data loads or stores. Instruction fetches include fetches for sequential execution, speculative and nonspeculative branch targets, prefetches, and interrupts.

The Address Range Registers, described in *Section 6.3.1.3* on page 155, are used together with the `RclassIDs` defined in the following registers (two pairs, one for each PPE thread) to identify addresses covered by the RMT:

- Instruction Class ID Register 0 (`ICIDR0`).
- Instruction Class ID Register 1 (`ICIDR1`).
- Data Class ID Register 0 (`DCIDR0`).
- Data Class ID Register 1 (`DCIDR1`).

Cache management instructions (*Section 6.1.6* on page 146) are treated like load or store instructions with respect to `RClassID`.

6.3.1.2 *Index and Table Structure*

The PPE's TLB RMT is implemented by the `PPE_TLB_RMT` register, and the 3-bit `RClassID` is used as an index into this 8-entry RMT. For each `RclassID`, the `PPE_TLB_RMT` defines which ways of the TLB are eligible to be replaced when a TLB miss occurs. The high-order 32 bits of this register are not implemented and are reserved for future use. The low-order 32 bits are divided into eight 4-bit RMT entries, corresponding to eight `RclassID`s.

Each of the eight 4-bit RMT entries has a one-to-one correspondence to the four ways in the TLB. For each RMT-entry bit with a value of '1', the table-lookup algorithm can replace an entry in that way if a TLB miss occurs in the EA range that maps to the `RclassID` for that RMT entry. If multiple ways are indicated, they are replaced in priority order beginning with invalid entries first, and then valid entries according to the pseudo-LRU policy described in *Section 4.3.5.2 Hardware TLB Replacement* on page 109.

For example, if a translation with an `RclassID` of 5 is requested, that class corresponds to the `RMT5` field of the `PPE_TLB_RMT` register, which has a value of '1001'. If a TLB miss occurs, TLB ways 0 and 3 are valid candidates for replacement (pseudo-LRU initially chooses way 0 and then points to way 3 when another miss occurs). If all bits in an RMT field are written by software with zero ('0000'), then the hardware treats this RMT field as if it had been set to '1111', therefore allowing replacement to any way. (This corresponds to the method used for SPE TLB replacement described in *Section 6.3.3* on page 158.)

The `RMT0` field of the `PPE_TLB_RMT` register is the default entry and should be used by software to specify the replacement policy for any EA range that is not mapped by the Address Range Registers.

6.3.1.3 *Address Range Registers*

The PPE has a set of Address Range Registers for supporting the pseudo-LRU TLB replacement algorithm (the registers are also used for L2 replacement). An Address Range is a naturally-aligned range that is a power-of-2 size between 4 KB and 4 GB, inclusive. An Address Range is defined by two types of registers; a Range Start Register (RSR) and a Range Mask Register

(RMR). For each Address Range, there is an associated ClassID Register (CIDR) that specifies the `RclassID`. The names RSR, RMR, and CIDR are generic labels for groups of registers that are differentiated according to Instructions or Data (I or D prefix).

As shown in *Table 6-2*, for each PPE thread there are two sets of RSRs, RMRs, and CIDRs for load/store data accesses and two sets for instruction fetches. The Address Range Registers are accessible by the **mtspr** and **mfspr** privileged instructions.

Table 6-2. Address Range Registers (One per PPE Thread)

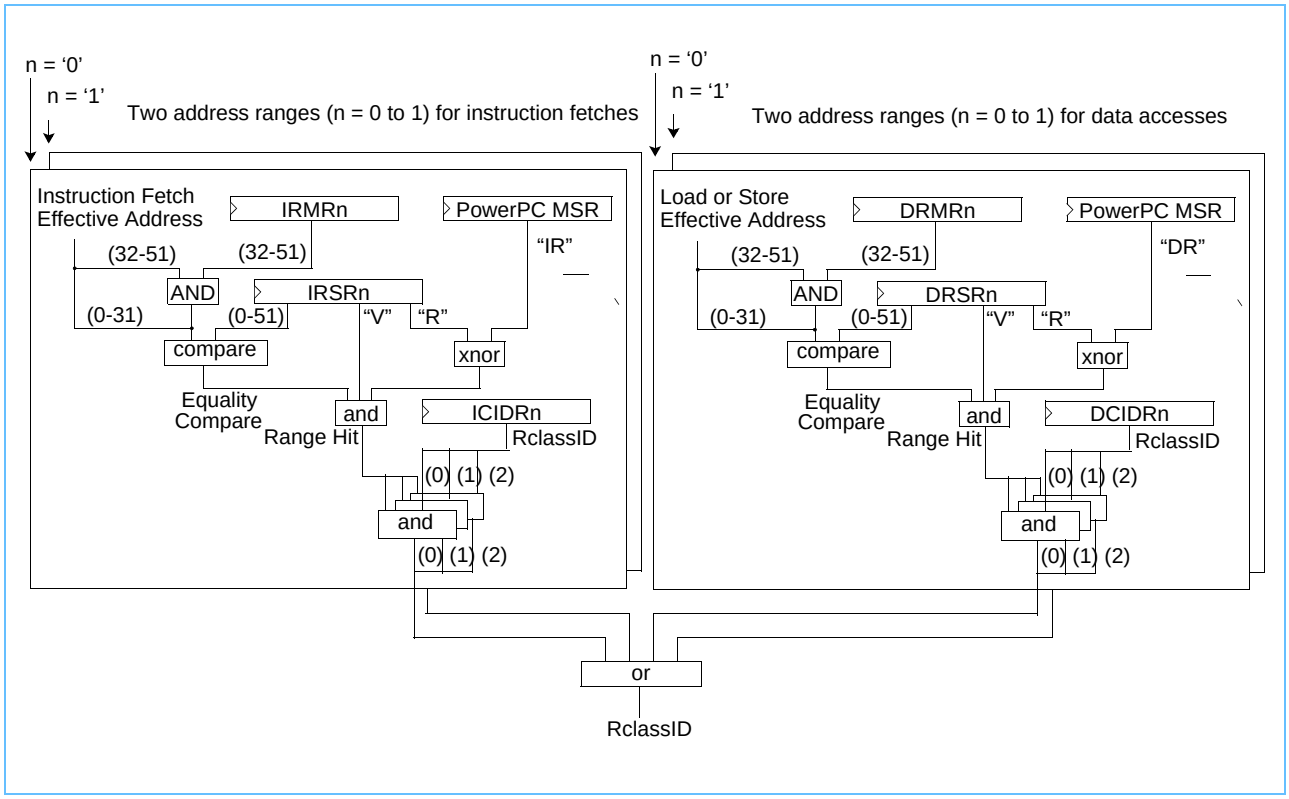
Type of Operation	Address Range		Replacement ClassID (`RclassID`)
	Range Start Registers	Range Mask Registers	
Data Access	Data Address Range Start Register 0 (DRSR0)	Data Address Mask Register 0 (DRMR0)	Data Class ID Register 0 (DCIDR0)
	Data Address Range Start Register 1 (DRSR1)	Data Address Mask Register 1 (DRMR1)	Data Class ID Register 1 (DCIDR1)
Instruction Fetch	Instruction Range Start Register 0 (IRSR0)	Instruction Range Mask Register 0 (IRMR0)	Instruction Class ID Register 0 (ICIDR0)
	Instruction Range Start Register 1 (IRSR1)	Instruction Range Mask Register 1 (IRMR1)	Instruction Class ID Register 1 (ICIDR1)

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

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

If there is no range hit for a given EA, the `RclassID` has a value of ‘0’. In effect, RMR defines the size of the range by selecting the bits of an EA used to compare with the RSR. The upper bits of an RSR contain the starting address of the range and the lower bits contain a relocation mode (real or virtual) and a valid bit. The size of the range must be a power-of-2. The starting address of the range must be a range-size boundary.

Figure 6-3. Generation of RclassID from the Address Range Registers for Each PPE Thread



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

6.3.2 PPE L2 Replacement Management Table

The PPE supports an 8-entry RMT for the 8-way set associative L2 cache. It is implemented by the L2 RMT Setup Register (`L2_RMT_Setup`). The L2 RMT uses the same Address Range Registers that are used by the TLB RMT (see *Section 6.3.1.3* on page 155). The L2 pseudo-LRU replacement algorithm, for which these registers are used, is described in *Section 6.1.5.3 Replacement Algorithm* on page 143. For each `RclassID`, the L2 RMT defines which ways of the L2 cache are eligible to be replaced when an L2-cache miss occurs.

The L2-cache replacement policy is controlled by a Replacement Management Table analogous to that used for the PPE TLB (*Section 6.3.1 PPE TLB Replacement Management Table* on page 154) when the L2 cache is not in direct-mapped mode (see *Section 6.1.5.3 Replacement Algorithm* on page 143). The RMT and LRU functions of the L2 cache can be configured in the L2 Mode Setup Register 1 (`L2_ModeSetup1`).

The workings of the L2 RMT are analogous to those of the TLB RMT, described above, except that the L2 cache has eight ways instead of four ways. The 3-bit `RclassID` is used as an index into the `L2_RMT_Setup` register to select one of the RMT entries. Each entry of the RMT contains a replacement-enable bit for each of the eight ways in the L2 cache. The `RclassID` selects one of the eight fields in this register.

Each of the eight bits of an RMT entry has a one-to-one correspondence to the eight ways in the L2 cache. For each RMT-entry bit with a value of '1', the L2 replacement algorithm can replace an entry in that way if a miss occurs in the EA range that maps to the `RclassID` for that RMT entry. A bit value of '0' indicates that the entry is locked and cannot be replaced. If multiple ways are indicated, they are replaced in priority order beginning with invalid entries first, and then valid entries. If all bits in an RMT field are zero ('0000 0000'), then the hardware treats this RMT field as if it had been set to '1111 1111', therefore allowing replacement to any way. The replacement method is analogous to that used by the pseudo-LRU policy for TLB replacement, described in *Section 4.3.5.2 Hardware TLB Replacement* on page 109.

The L2 RMT can be used by software to achieve various results, including:

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

6.3.3 SPE TLB Replacement Management Table

The Synergistic Memory Management (SMM) unit of each SPE supports a 4-entry RMT that can be used to lock TLB entries and prevent their replacement. The SPE's TLB RMT is analogous to that of the PPE's, described in *Section 6.3.1 PPE TLB Replacement Management Table* on page 154.

The SPE RMT is enabled with the `SMM_HID[RMT_Dsbl]` bit. The RMT entries are stored in the MFC TLB Replacement Management Table Data Register (`MFC_TLB_RMT_Data`)[14]. A 2-bit `RClassID` is provided as a parameter in MFC commands. RMT-entry selection is specified by the `RClassID` given to the SMM by the DMAC during a translation request. The `RClassID` is defined in the MFC Class ID Command Opcode (`MFC_ClassID_Cmd`) register; unlike the PPE RMTs, the SPE RMT does not use the system of Address Range Registers described in *Section 6.3.1.3 Address Range Registers* on page 155.

14. The CBE processor does not implement the RMT Data Register (RMT_Data) defined in the *Cell Broadband Engine Architecture*. Instead, the SMM uses the implementation-dependent MFC TLB Replacement Management Table Data Register (MFC_TLB_RMT_Data).

Each 4-bit RMT entry represents one of the four TLB ways in a congruence class (see *Figure 4-6 SMM Virtual- to Real-Address Mapping* on page 108 for an illustration of the TLB structure). A bit value of ‘1’ indicates that the corresponding TLB way can be replaced. A bit value of ‘0’ indicates that the way is locked and cannot be replaced. If an entire RMT entry is enabled (‘1111’), then all TLB ways in the congruence class represented by the RMT entry are available for replacement, and the LRU algorithm chooses the replacement set, as described in *Section TLB Replacement Policy* on page 110. If all bits in an RMT field are written by software with zero (‘0000’), then the hardware treats this RMT field as if it had been set to ‘1111’, therefore allowing replacement to any way. This corresponds to the method used for PPE TLB replacement described in *Section 6.3.1.2* on page 155.

6.4 I/O Address-Translation Caches

For a description of the caches used in I/O address-translation, and their management, see *Section 7.4 I/O-Address Translation* on page 176 and *Section 7.6 I/O Address-Translation Caches* on page 182.

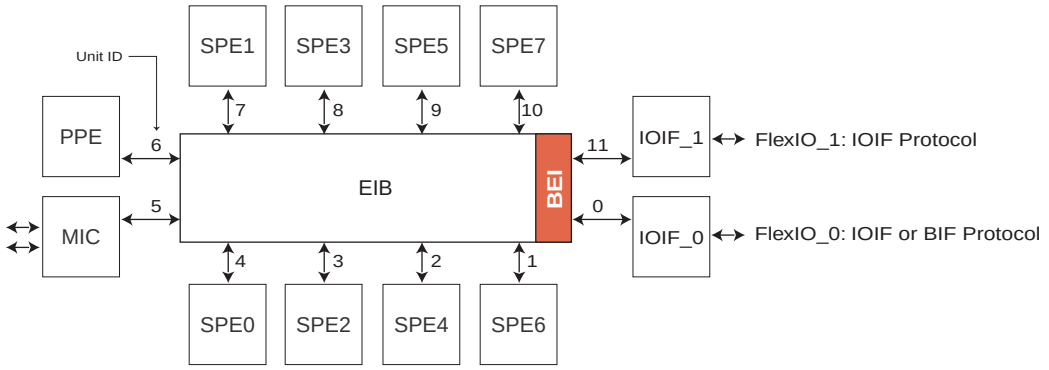
7. I/O Architecture

7.1 Overview

7.1.1 I/O Interfaces

The Cell Broadband Engine Interface (BEI) Unit, shown in *Figure 7-1*, manages data transfers between the processor elements on the Element Interconnect Bus (EIB) and I/O devices. The BEI supports two Rambus FlexIO I/O bus interfaces, FlexIO_0 and FlexIO_1. These bus interfaces are also called IOIF0 and IOIF1, respectively.

Figure 7-1. BEI and its Two I/O Interfaces



BEI	Cell Broadband Engine Interface Unit	IOIF	I/O Interface, or I/O Interface Protocol
BIF	Cell Broadband Engine Interface Protocol	MIC	Memory Interface Controller
EIB	Element Interconnect Bus	PPE	PowerPC Processor Element
FlexIO	Rambus FlexIO Bus	SPE	Synergistic Processor Element

The input and output bandwidth (number of transmitters and receivers) of each I/O interface (IOIF) is software-configureable during the CBE processor's Power-On Reset (POR) sequence, described in the *Cell Broadband Engine Hardware Initialization Guide*. The FlexIO_0 interface can be configured at higher input and output bandwidths than the FlexIO_1 interface. The FlexIO_0 interface can also be configured with either the non-coherent I/O Interface (IOIF) protocol or the fully coherent Cell Broadband Engine Interface (BIF) protocol, which is the EIB's internal protocol. The FlexIO_1 interface supports only the non-coherent IOIF protocol. Each IOIF-protocol interface has four virtual channels.

Both the IOIF and BIF protocols provide fully pipelined, packet-transaction interconnection using credit-based flow control. The non-coherent IOIF protocol can interface to I/O devices, non-coherent memory subsystems, switches, or bridges. The coherent BIF protocol, which supports memory coherency and data synchronization, can be connect to another CBE processor or coherent memory subsystem, as well as a compatible switch or bridge.

The POR configuration bits are not visible to CBE software after POR—except in system designs in which the external System Controller chip makes them visible—but several related parameters of the configuration can be modified by CBE software after initialization[15].

The BEI has an I/O interface controller (IOC) that supports the two IOIFs with I/O-command processing, I/O-address translation, I/O-interrupt control, programmable memory-mapped I/O (MMIO) registers, and, in general, implements the I/O facilities. Both IOIFs use the same bus address space—a 42-bit real-address space. The BEI also adds a paging mechanism for addresses from I/O devices that is independent of the standard *PowerPC Architecture* memory-paging mechanism. The PPE controls the system-management resources used for both memory paging and I/O paging. It is possible to extend a current kernel's I/O architecture to support this addition of I/O paging without having to re-architect the kernel.

Use of the IOIFs can be controlled using tokens that are allocated by the CBE processor's Resource Allocation Manager (RAM), as described in *Section 8 Resource Allocation Management* on page 203.

7.1.2 System Configurations

Figure 7-2 on page 163 shows two possible system configurations. At the top is a system with a single CBE processor that connects to coherent I/O devices and non-coherent I/O devices. At the bottom is a system with two CBEs that connect together, forming a coherent symmetric multiprocessor (SMP) system, with each CBE processor also connecting to non-coherent I/O devices.

Figure 7-3 on page 164 shows a four-way configuration, which requires a BIF-protocol switch to connect the four CBEs. This figure does not necessarily represent future system-product plans.

15. The coherency protocol for the FlexIO_0 interface is software-selectable only at Power-On Reset (POR). However, if an I/O interface is known to use the IOIF protocol, an I/O device on that interface can access memory pages coherently by setting the M bit in an I/O Page Table (IOPT) entry.

Figure 7-2. Single and Dual CBE System Configurations

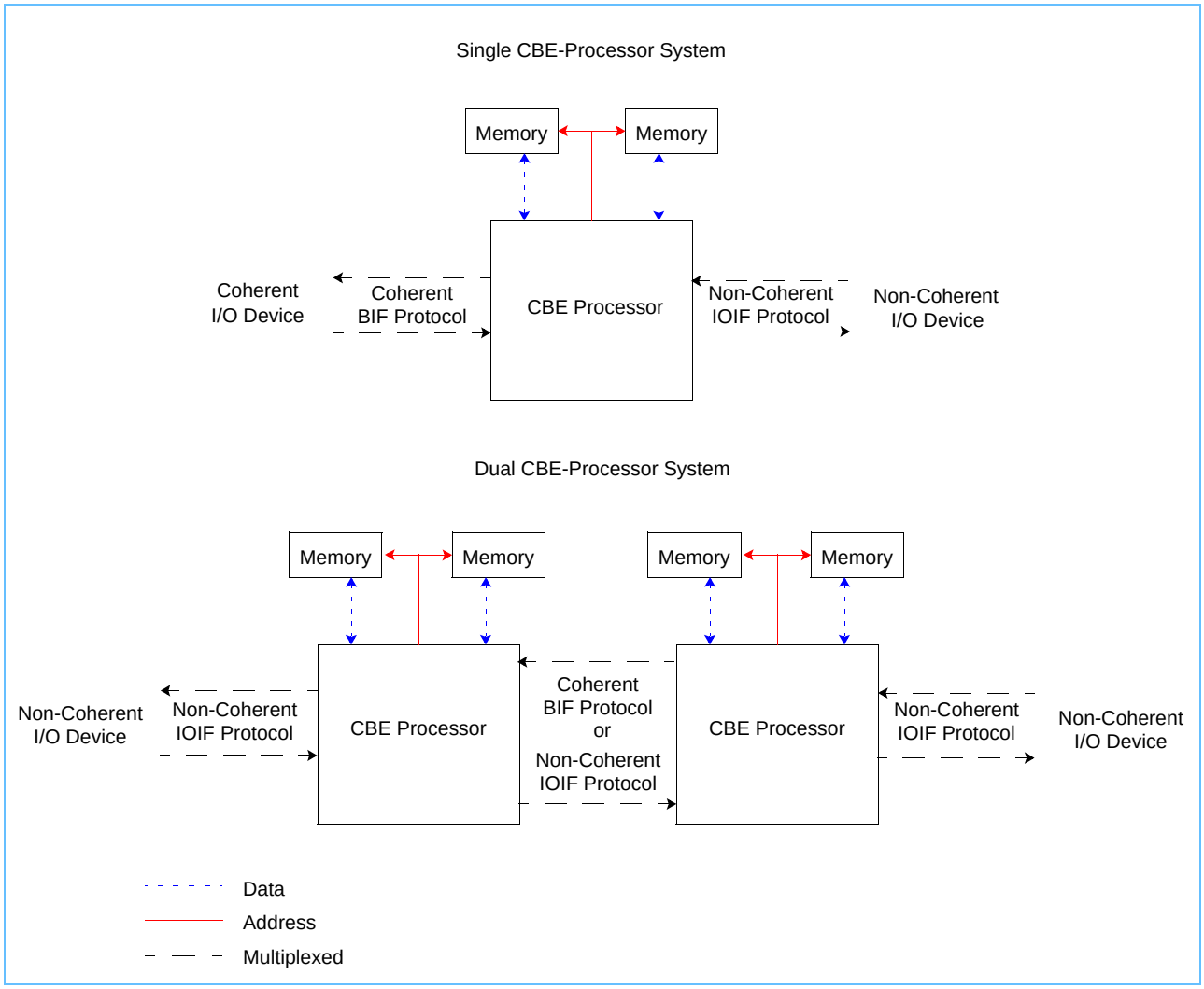
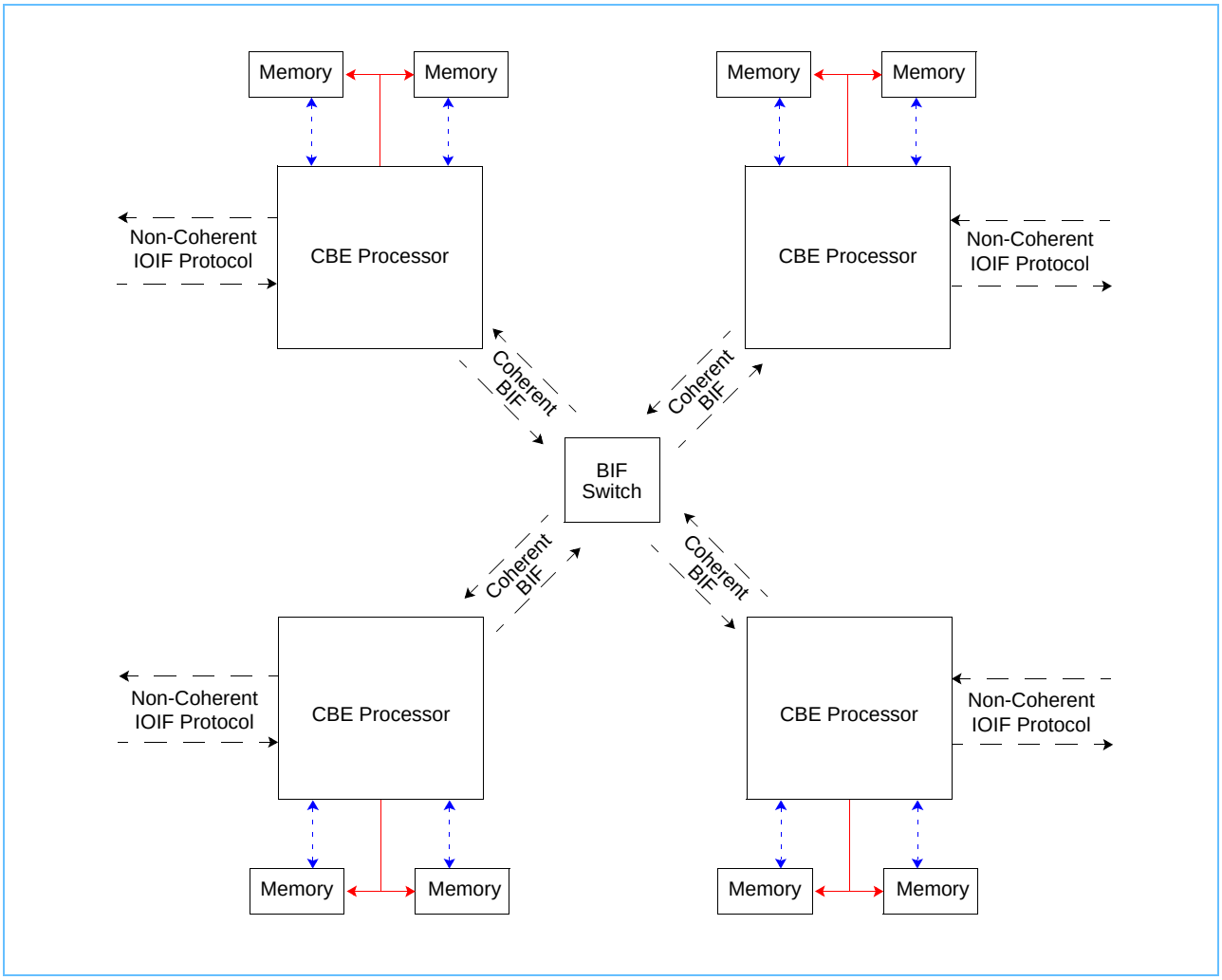


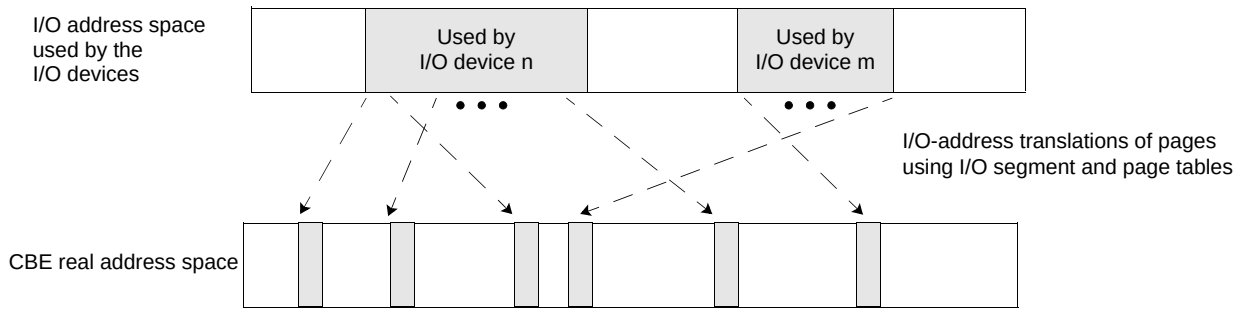
Figure 7-3. Quad CBE System Configuration



7.1.3 I/O Addressing

The CBE processor supports I/O-address translation on interfaces configured for the IOIF protocol (but not for the BIF protocol). An I/O address is an address passed from an I/O device to the CBE processor when the I/O device attempts to perform an access to the CBE processor's real-address space—for example, to an MMIO register, to the LS of an SPE, or to the memory attached to the CBE processor. For I/O accesses coming into the CBE processor on an I/O interface, the CBE processor provides two modes for handling the incoming address: if I/O-address translation is disabled, the I/O address is used as a real address. If I/O-address translation is enabled, the address is converted from an I/O address into a real address, as shown in *Figure 7-4* on page 165.

Figure 7-4. Use of I/O Addresses by an I/O Device



I/O operations often involve multiple, non-contiguous real pages. The I/O address-translation mechanism supports the use of a contiguous I/O address range assigned to non-contiguous real pages. Each I/O device can be assigned a separate range of I/O addresses, either statically or dynamically.

I/O devices can only access pages that have been set up in an I/O page table with their I/O identifier (IOID). This identifier must be passed on the I/O interface (IOIF) bus to the CBE processor, along with the I/O address, during the I/O access. If the identifier from the I/O interface does not match the value in the CBE processor's I/O page table entry corresponding to the I/O address, an I/O exception occurs. Thus, a form of storage protection is enforced. Pages can also be protected according to their read/write attributes, and pages can be assigned memory-coherence and storage-order attributes.

In some systems, such rigid protection between all I/O devices may not be required. For example, if all I/O devices on a certain I/O bus in the I/O subsystem are only used by one logical partition, it may be satisfactory for the I/O subsystem to group these I/O devices into one identifier.

7.2 Data and Access Types

7.2.1 Data Lengths and Alignments

The following data lengths and alignments are supported for read and write transfers across an IOIF:

- Read and write transfers of 1, 2, 4, 8, and 16 bytes that are naturally aligned. (An I/O device, but not the CBE processor, may also perform read and write transfers of 3, 5, 7, 9, 10, 11, 12, 13, 14, and 15 bytes, if they do not cross a 16-byte boundary.)
- Read and write transfers of q-times-16 bytes, where q is an integer from 1 to 8, that are quadword-aligned and do not cross a 128-byte boundary.

SPEs and I/O devices can perform DMA transfers of data lengths and alignments that differ from the above by splitting the accesses into multiple transfers across the IOIF.

7.2.2 Atomic Accesses

Atomic IOIF accesses are supported for naturally aligned 1, 2, 4, and 16-bytes transfers. These read and write transfers are fully serialized and complete before the next transfer is initiated. For example, suppose a naturally aligned 2-byte memory location initially contains x'0000'. If an I/O device writes x'1111' to this location, a PPE or SPE doing a half-word read of this location will load either x'0000' or x'1111'. It will never load x'0011' or x'1100'.

7.3 Registers and Data Structures

The IOC supports several registers and data structures used for I/O-address translation (*Section 7.4 I/O-Address Translation* on page 176) and related purposes. The bit fields and values for these registers are described in the *Cell Broadband Engine Registers* specification, although in some cases the descriptions in this section have additional detail.

7.3.1 IOCmd Configuration Register

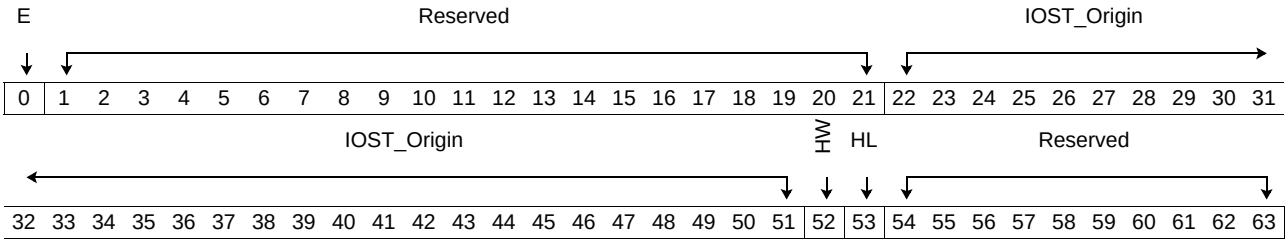
The IOCmd Configuration Register (`IOC_IOCmd_Cfg`) configures basic settings for the IOC. These settings include the I/O identifier (IOID) for each I/O interface, I/O address-translation enable, token parameters, timeout, read intervention, and others.

7.3.2 I/O Segment Table Origin Register

The 64-bit I/O Segment Table Origin (`IOC_IOST_Origin`) register enables I/O-address translation, specifies the origin and size of the I/O Segment Table (IOST), specifies the hint-or-lock function of the hint bit in IOST and I/O Page Table (IOPT) entries, and enables hardware cache-miss handling for the IOST and IOPT caches.

The `IOC_IOST_Origin` register should be initially configured at boot time, before there are any I/O accesses, and it can be modified after boot. If a hypervisor is implemented (see *Section 11* on page 327), software would typically map the `IOC_IOST_Origin` register using a page that is only accessible to the hypervisor or firmware.

7.3.2.1 *Register Layout and Fields*



Bits	Field Name	Description
0	E	I/O-Address Translation Enable: 0 Disabled (default). I/O addresses are treated as real addresses, and the I/O-address translation unit is disabled to save power. The only IOC Address Translation MMIO register that can be accessed is this register. 1 Enabled. I/O addresses are translated using the IOST and IOPT(s). The IOC_IOCmd_Cfg[TE] bit must be set to the same value as this E bit.
1:21	Reserved	Bits are not implemented; all bits read back as ‘0’.
22:51	IOST Origin	The real page number (RPN) of the IOST. The IOST must be integrally aligned and must be aligned on a 4-KB page. These are the 30 high-order bits of the 42-bit real address of the IOST.
52	HW	Hardware Cache-Miss Handling Enable for IOST and IOPT caches: 0 Hardware miss handling is disabled. 1 Hardware miss handling is enabled.
53	HL	Hint or Lock for IOST and IOPT caches: 0 The hint bit for an IOST cache entry or IOPT cache entry is treated as just a hint for valid entries. If there are no invalid entries in the applicable congruence class, hardware miss handling should replace an entry whose hint bit is '0' instead of an entry whose hint bit is '1'. If there are no invalid entries in the congruence class and all entries in the congruence class have hint bits equal to '1', hardware miss handling will replace a cache entry whose hint bit is '1'. 1 The hint bit is treated as a lock. If hardware miss handling is enabled and an IOST or IOPT cache miss occurs, hardware must not replace a cache entry whose hint bit is '1', even if the valid (V) bit in the entry is '0'.
54:57	Reserved	
58:63	IOST Size	This is the base-2 logarithm of the number of entries in the IOST. The first implementation of the *CBEA* (the CBE processor) treats this field as a fixed value of 7, so that the IOST has 128 entries.

7.3.2.2 ***Register Functions***

I/O-Address Translation

The `IOC_IOST_Origin` register’s E bit enables I/O-address translation on interfaces configured for the IOIF (but not BIF) protocol. It determines whether addresses generated by I/O units will be used as real addresses (untranslated) or whether the translation mechanism will produce the real address based on the IOST and IOPT. The primary advantages of enabling I/O-address translation include:

- Identity-matching of the I/O identifier (IOID) and the real address in the IOPT.
- Read/write protection of the main-storage space during accesses by I/O units.
- Memory coherence and storage ordering of I/O accesses.
- Hardware control of misses in the IOST and IOPT caches.
- Sizing of the IOST and its pages.

Unlike PowerPC virtual-memory address translation, the use of I/O-address translation does not include access to a larger address space; in fact, the translated address space is smaller (2^{35} bytes) than the untranslated real-address space (2^{42} bytes).

IOST Origin

The IOST_Origin field is architected to contain up to 50 high-order bits of the 62-bit real address of the IOST. However, the first implementation of the *CBEA* (the CBE processor) supports 42-bit real addresses, and only the right-most (low-order) 30 bits (bits 22:51) of the IOST_Origin field are used; the upper 20 bits of the field are reserved and treated as ‘0’s.

Hardware Cache-Miss Handling

The hardware cache-miss (HW) bit enables or disables hardware handling of misses in the IOST cache (*Section 7.6.1* on page 182) and IOPT cache (*Section 7.6.2* on page 184). When HW = ‘1’, the hint or lock (HL) bit indicates to hardware whether IOPT entries with their hint (H) bit = '1' (*Section 7.3.4* on page 171) can be replaced.

If HL = '1', hardware miss handling avoids loading an IOPT-cache entry whose H = '1' in the IOPT. In this case, hardware will select an entry using the following criteria order if there is a cache miss:

1. An entry in the applicable congruence class (cache index) whose valid (V) bit in the IOPT and hint (H) bit are '0'.
2. An entry in the applicable congruence class whose hint (H) bit is '0'. A pseudo-LRU algorithm is used in conjunction with this criterion to select an entry among those that meet the criteria.

If no entry is found based on the preceding criteria, and if a cache miss occurs in that congruence class, an I/O segment fault (*Section 9.6.4.4* on page 271) occurs, instead of an I/O page fault, even if the entry exists in the IOST. Likewise, if all the H bits in the applicable congruence class of an IOPT cache are '1', and if a cache miss occurs in that congruence class, an I/O page fault (*Section 9.6.4.4* on page 271) occurs even if the entry exists in the IOPT.

If HL = '0', hardware miss handling will select an entry using the following criteria order:

1. An entry in the applicable congruence class whose V bit is '0'.
2. An entry in the applicable congruence class whose H bit is '0'.
3. An entry in the applicable congruence class whose H bit is '1'.

A pseudo-LRU algorithm is used in conjunction with criteria 2 and 3 to select an entry among those that meet the criteria.

See *Section 7.6.1.3* on page 184 for further details about hardware and software cache-miss handling.

IOST Size

The IOST Size field in the first implementation of the *CBEA* (the CBE processor) is treated by the hardware as if the value is always 7. Thus, the IOST has 128 entries. The corresponding bits in the `IOC_IOST_Origin` are not implemented.

7.3.2.3 *Register Modifications*

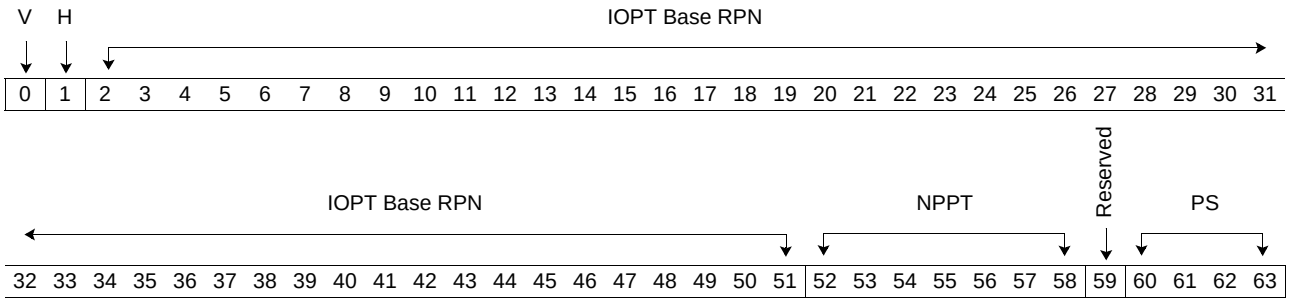
Software must ensure that there are no I/O accesses occurring when it modifies the `IOC_IOST_Origin` register. Also, the `IOC_IOCmd_Cfg[TE]` and `IOC_IOST_Origin[E]` bits must both be set to the same value.

To guarantee that the new value of a store to `IOC_IOST_Origin` register is used on future I/O-address translations, software should load from the `IOC_IOST_Origin` register and ensure that the load has completed before a subsequent store that might cause the new `IOC_IOST_Origin` value to be used. A synchronization instruction or command (*Section 20* on page 551) that orders a load from storage that is both caching-inhibited and guarded versus the subsequent store can be used for this purpose.

7.3.3 I/O Segment Table

The I/O Segment Table (IOST) is a variable-sized data structure in memory that defines the first phase of the translation of an I/O address to a real address. For each I/O segment, the table contains an entry that points to a page table that is used to complete the translation. There are 2^7 segments, each one containing 2^{28} bytes, for a total translated address space of 2^{35} bytes.

7.3.3.1 *Table-Entry Layout and Fields*



Bits	Field Name	Description
0	V	Valid 0 The IOST entry is not valid. 1 The IOST entry is valid.
1	H	Hint 1 *Not valid in first implementation of the CBEA (the CBE processor).* (Architecturally, if IOC_IOST_Origin[HL] = '0', this is a hint to the IOST cache to retain this entry if possible. If IOC_IOST_Origin[HL] = '1', this entry is never displaced from an IOST cache by hardware when handling an IOST cache miss.) 0 *The IOST hint bit is treated as a '0' in the first implementation of the CBEA (the CBE processor).* No cache hint.
2:51	IOPT Base RPN	IOPT-base real page number (RPN) This is the RPN of the first entry in the IOPT for this I/O segment. Only the right-most 30 bits of the IOPT Base RPN are implemented. The upper 20 bits of the IOPT Base RPN are treated as '0'es.
52:58	NPPT	Number of 4-KB Pages (minus one) in the Page Table for this I/O segment. For IOPT format 1 (*Section 7.3.4* on page 171), the number of IOPT entries for the segment is equal to 512 times the sum of the NPPT plus 1. The IOPT entries are for the lowest-numbered pages in the segment. (IOPT format 2 is not supported in first implementation of the *CBEA* (the CBE processor).
59	Reserved	
60:63	PS	Page size of the pages in this I/O segment. The page size is 4^{PS} KB. PS values are: • 0001 = 4 KB page size • 0011 = 64 KB page size • 0101 = 1 MB page size • 0111 = 16 MB page size All four page sizes can be used, regardless of the page sizes used by the PPE and SPEs.

7.3.3.2 *Table-Entry Functions*

IOST entries specify the properties of a segment and its page tables, including the validity of the IOST entry, the real page number (RPN) of the first page table in this I/O segment, the number of pages in a page table, and the page size.

The IOST can contain up to 128 64-bit entries. The IOST must start on a 4-KB boundary, at a minimum, and it must be naturally aligned.

Validity

Software can control access to pages in a segment by writing the IOST entry's valid (V) bit. If an I/O translation attempts to use an IOST entry that has V = '0', an I/O segment-fault type of I/O exception occurs.

Cache Hint

In the first implementation of the *CBEA* (the CBE processor), the IOST hint (H) bit is always treated as a '0'. Thus, hints or locks for the IOST cache (*Section 7.6.1* on page 182) are not supported, irrespective of the value of the `IOC_IOST_Origin[HL]` bit.

Page Size

The IOST page-size (PS) field defines the size of the pages in the I/O segment. Four page sizes are supported—4 KB, 64 KB, 1 MB, and 16 MB. However, since it is possible to disable I/O-address translation and use an external I/O address-translation mechanism, the page sizes supported by an external I/O-address translation mechanism could differ from those supported by the PPE and SPEs.

7.3.3.3 *Table-Entry Modifications*

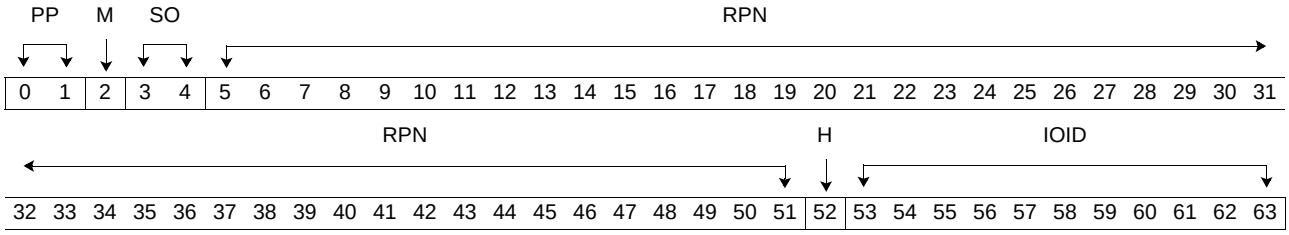
An IOST entry should only be modified when there is no simultaneous I/O access that uses the IOST entry; if this condition is not met, it is unpredictable whether the I/O access will use the old or new value of the IOST entry. If software chooses to stop simultaneous I/O accesses, such accesses must be stopped at the I/O unit. To prevent an I/O-address-translation access of the IOST from getting part of an old value and part of a new value when there is a simultaneous I/O access using the entry, software should do either of the following:

- Use a doubleword atomic store to the IOST entry. According to *PowerPC Architecture*, this is a single-register doubleword store that is naturally aligned.
- Use an atomic store to set the IOST valid (V) bit to ‘0’, execute an **eieio**, **lwsync**, or **sync** instruction, and then store to other bytes in the IOST entry. After making the updates, execute an **eieio**, **lwsync**, or **sync** instruction, and then execute an atomic store to set the IOST V bit to ‘1’. The **lwsync** instruction can only be used for this purpose if the IOST is in storage that is not caching-inhibited. The **eieio** and **sync** instructions are unnecessary if the IOST is in storage that is both caching-inhibited and guarded.

7.3.4 I/O Page Table

The I/O Page Table (IOPT) is a variable-sized data structure in memory. This table defines the second phase of the translation of an I/O address to a real address. For each I/O page, the table may contain an entry that specifies the corresponding real page number. The IOPT must start on a 4-KB boundary.

7.3.4.1 *Table-Entry Layout and Fields (Format 1)*



Bits	Field Name	Description
0:1	PP	Page Protection 00 No access. (This can also be used to designate an invalid entry.) 01 Read access only. 10 Write access only. 11 Read and write access.
2	M	Coherence Required 0 Memory coherence not required. 1 Memory coherence required.
3:4	SO	Storage Ordering for accesses performed on one IOIF with the same IOID and virtual channel. 00 Previous reads and writes are not necessarily performed before a write. 01 Reserved. 10 Previous writes are performed before a write. 11 Previous reads and writes are performed before a read or write.
5:51	RPN	Real Page Number of the translated I/O address: • For 64-KB page sizes, the 4 least-significant bits of the RPN must be cleared to ‘0’. • For 1-MB page sizes, the 8 least-significant bits of the RPN must be cleared to ‘0’. • For 16-MB page sizes, the 12 least-significant bits of the RPN must be cleared to ‘0’. The real address supported has only 42 bits; therefore, bits[0:16] of the RPN field must be ‘0’s in IOPT entry format 1.
52	H	Hint 1 If IOC_IOST_Origin[HL] = '0', this is a hint to the IOPT cache to retain this entry if possible. If IOC_IOST_Origin[HL] = '1', this entry is never displaced from an IOPT cache by hardware when handling an IOPT cache miss. 0 No cache hint.
53:63	IOID	I/O Device Identifier. Only the I/O device specified can have its I/O address translated using this IOPT entry.

7.3.4.2 *Table-Entry Functions*

The entries in this table specify the properties of an I/O page, including its read-write protection, memory coherence, storage-ordering, real page number (RPN) of the page’s translated I/O address, hint for entry-caching, and the I/O Device Identifier (IOID) of the device(s) to which this page applies. This format is called Format 1 for IOPTs; it is the only format supported in the first implementation of the *CBEA* (the CBE processor).

Generally, software should not create IOPT entries that give I/O devices access to the IOST or IOPT, since this would allow the I/O device to circumvent I/O-address translation storage protection.

Page Protection

The page protection (PP) bits in the IOPT entry define the type of access allowed for the I/O unit. If PP = ‘00’, no access is allowed. If PP = ‘01’ or ‘11’, an I/O read of the page is permitted. If PP = ‘10’ or ‘11’, an I/O write of the page is permitted. If the access is attempted but not permitted, an I/O page-fault type of I/O exception occurs. Software can set PP = ‘00’ to mark invalid entries.

Memory Coherence

The IOPT M bit is similar to the PowerPC memory-coherence-required storage attribute. If M = ‘1’ and the IOIF C bit = ‘1’ (*Section 7.7.1* on page 189), memory coherence is maintained by hardware. If M = ‘0’ or if the IOIF C bit = ‘0’, memory coherence is not maintained by hardware. For storage that does not have the memory-coherence-required attribute, software must explicitly manage memory coherence to the extent required for the program to run correctly. Refer to *PowerPC Virtual Environment Architecture, Book II* for a detailed definition of the memory-coherence-required storage attribute.

Storage Order

The SO bits define the storage order rules that apply to reads and writes from one IOIF with the same virtual channel and same IOID, unless the IOIF S bit (*Section 7.7.2.2* on page 191) allows a more relaxed ordering. Previous writes on the same IOIF with the same virtual channel and the same IOID will be performed before a write with a corresponding SO = ‘10’ or ‘11’ and with the IOIF S bit = ‘1’. Previous reads and writes on the same IOIF with the same virtual channel and same IOID will be performed before a read or write with a corresponding SO = ‘11’ and with the IOIF S bit = ‘1’.

If SO = '00' or the IOIF S bit = ‘0’, the CBE processor does not order the access relative to previous accesses. See *Section 7.7.2.2* on page 191 for additional details, including complexities associated with locations that do not have the memory-coherence-required storage attribute.

Real Page Number

An I/O access will read or write the real page number (RPN) of the translated I/O address if the page-protection (PP) bits allow the access for the specified I/O device. If 2^p bytes is the page size (as specified in the IOST PS field, *Section 7.3.3* on page 169), and p is greater than 12 (page size greater than 4 KB), then the least-significant (p - 12) bits of the RPN must be ‘0’. The first implementation of the *CBEA* (the CBE processor) supports 42 real address bits; bits 0 through 16 of the RPN field must be ‘0’s in IOPT entry. The I/O address is checked to ensure that it does not access a page beyond the number of pages in the segment, as defined by the NPPT field of the IOST.

Assume a page size of 2^p bytes is specified in the IOST PS field. If the value of the most-significant (28 - p) bits of the 28 least-significant I/O address bits is greater than 512 times the NPPT, an I/O page fault (*Section 9.6.4.4* on page 271) occurs.

Cache Hint

Depending on the `IOC_IOST_Origin[HL]` bit (*Section 7.3.2* on page 166), the IOPT hint (H) bit is treated as either a hint or a lock. If H = '1', it is a hint to the hardware to retain a valid IOPT-cache entry, if possible; if H = '1' and `IOC_IOST_Origin[HL]` = ’1’, the IOPT entry is never displaced by hardware during an IOPT cache miss.

The IOPT cache (*Section 7.6.2* on page 184) is 4-way associative with 64 congruence classes. For a given congruence class, if no valid IOPT entries have H = ‘1’ and `IOC_IOST_Origin[HW]` = ‘1’, then an IOPT-cache miss in this congruence class causes hardware to select one of the four ways in the congruence class, based on a pseudo-least-recently-used (LRU) algorithm. For an IOPT-cache miss in a second congruence class with two valid entries having H = ‘1’ and two

entries having H = ‘0’, the IOPT entry will be placed in one of the cache entries having H = ‘0’. For an IOPT-cache miss in a third class with three valid entries having H = ‘1’ and one entry having H = ‘0’, the IOPT entry will be placed in the cache way with the entry having H = ‘0’.

I/O Device Identifier

The IOID bits specify the I/O device identifier. Inbound accesses from an I/O unit to the EIB contain an IOID as part of their access command, and hardware compares this access-command IOID to the IOID in the IOPT. Only the I/O device with this IOID can have its I/O address translated using the IOPT entry. If other I/O devices attempt translation using this IOPT entry, it will result in an I/O exception.

Looped-Back Operations

The first implementation of the *CBEA* (the CBE processor) does not support looped-back operations from an I/O device.

Architecturally, however, support for looped-back operations is optional for each implementation of the *CBEA*. When supported, commands that are received by the CBE processor from an IOIF device that are routed back out the same IOIF are called *looped-back operations*. To prevent IOIF looped-back operations when I/O-address translation is enabled, one option is to ensure that the RPNs in the IOPT are not in the range of real addresses mapped to an IOIF. If direct accesses from one IOIF to another IOIF are desired, the RPN(s) corresponding to the second IOIF need to be in the IOPT. In this case, looped-back operations from this second IOIF can be prevented by using IOIDs for the IOIF-to-IOIF accesses that are different from IOIDs used on the second IOIF.

7.3.4.3 ***Table-Entry Modifications***

An IOPT entry should only be modified when there is no simultaneous I/O access that uses the IOPT entry; if this condition is not met, it is unpredictable whether the I/O access will use the old or new value of the IOPT entry. If software chooses to stop simultaneous I/O accesses, such accesses must be stopped at the I/O unit. To prevent an I/O-address translation access of the IOPT from getting part of an old value and part of a new value when there is a simultaneous I/O access using the entry, software should do either of the following.

- Use a doubleword atomic store to the IOPT entry. According to *PowerPC Architecture*, this is a single-register doubleword store that is naturally aligned.
- Use an atomic store to set the IOPT valid (V) bit to ‘0’, execute an **eieio**, **lwsync**, or **sync** instruction, and then store to other bytes in the IOPT entry. After making the updates, execute an **eieio**, **lwsync**, or **sync** instruction, and then execute an atomic store to set the IOPT V bit to ‘1’. The **lwsync** instruction can only be used for this purpose if the IOPT is in storage that is not caching-inhibited (but see *Section 7.7.1* on page 189). The **eieio** and **sync** instructions are unnecessary if the IOPT is in storage that is both caching-inhibited and guarded.
- Invalidate the IOPT cache to ensure that the new IOPT entry is read into the cache.

7.3.5 IOC Base Address Registers

The IOC accepts commands from the EIB in which the address of the command matches the real-address ranges specified for each IOIF in the following MMIO registers:

- IOC Base Address Register 0 (`IOC_BaseAddr0`).
- IOC Base Address Register 1 (`IOC_BaseAddr1`).
- IOC Base Address Mask Register 0 (`IOC_BaseAddrMask0`).
- IOC Base Address Mask Register 1 (`IOC_BaseAddrMask1`).

The `IOC_BaseAddr`*n* and `IOC_BaseAddrMask`*n* together identify the range of real addresses to be routed to IOIF*n*, where the value of *n* is ‘0’ or ‘1’ and is equal to the number of the I/O Interface. The IOC supports two IOIF buses, two `IOC_BaseAddr` registers, and two `IOC_BaseAddrMask` registers.

Since the CBE processor supports only 42-bits of real address, only the least-significant 11 bits of the `IOC_BaseAddr`*n* registers’ Base Real Address and Base Replacement Address fields are implemented. Likewise, only the least-significant 11 bits of the `IOC_BaseAddrMask`*n* registers’ mask fields are implemented.

Valid values in the mask field and the corresponding address range routed to an IOIF are shown in the *Table 7-1*.

Table 7-1. IOIF Mask and Size of Address Range

Mask in `IOC_BaseAddrMask`*n* Registers	Size of Address Range
x‘000007FF’	2 GB
x‘000007FE’	4 GB
x‘000007FC’	8 GB
x‘000007F8’	16 GB
x‘000007F0’	32 GB
x‘000007E0’	64 GB
x‘000007C0’	128 GB
x‘00000780’	256 GB
x‘00000700’	512 GB
x‘00000600’	1 TB
x‘00000400’	2 TB
x‘00000000’	4 TB

When modifying `IOC_BaseAddr`*n* or `IOC_BaseAddrMask`*n*, software should ensure that no accesses to the previous or new address range occur by using the following procedure:

1. Ensure that pages for the old and new address ranges have the guarded storage attribute.
2. Ensure that SPE accesses do not occur while these registers are modified, for example, by suspending MFC DMA transfers using the `MFC_CNTL` register.
3. Ensure that previous PPE accesses are performed before storing to `IOC_BaseAddr`*n* or `IOC_BaseAddrMask`*n* by using one of the synchronization facilities described in *Section 20* on page 551 (for example, the **sync** instruction).

4. Ensure that read and write commands from IOIF*n* are performed on the EIB before storing to `IOC_BaseAddr`*n* or `IOC_BaseAddrMask`*n*. One potential option is to stop such accesses at the IOIF device or I/O device, if such a facility is provided (an I/O device, here, means a device that is connected to an IOIF indirectly by an IOIF device). If the IOIF device or I/O device supports a facility to send an interrupt after its last access, then reception of the external interrupt by the PPE is a guarantee that the previous command from the IOIF has been performed on the EIB. Although this does not guarantee the data transfer has completed, it is sufficient to ensure that the command phase has completed. Alternatively, if reads and writes from the IOIF are translated by the IOC (`IOC_IOCmd_Cfg[TE]` = '1' and `IOC_IOST_Origin`[0] = '1'), then the pages corresponding to the range specified by `IOC_BaseAddr`*n* and `IOC_BaseAddrMask`*n* can be invalidated in the IOST (*Section 7.3.3* on page 169) or IOPT (*Section 7.3.4* on page 171) and IOST cache (*Section 7.6.1* on page 182) or IOPT cache (*Section 7.6.2* on page 184), and, if the IOIF device or I/O device supports a facility to send an interrupt after this invalidation, reception of this external interrupt by the PPE is a guarantee that the previous command from the IOIF has been performed on the EIB.
5. Disable accesses to the IOIF by setting `IOC_BaseAddrMask`*n*[E] = '0'.
6. Store to `IOC_BaseAddr`*n* with the new value.
7. Store to `IOC_BaseAddrMask`*n* with the new value, setting `IOC_BaseAddrMask`*n*[E] = '1'.
8. Load from `IOC_BaseAddrMask`*n* to ensure the effect of the previous store to `IOC_BaseAddrMask`*n*.
9. Execute a **sync** instruction to ensure the completion of the previous load before subsequent accesses.

7.3.6 I/O Exception Status Register

The I/O Exception Status Register (`IOC_IO_ExcpStat`) captures error information for an I/O exception. This register is described in *Section 7.5 I/O Exceptions* on page 181.

7.4 I/O-Address Translation

7.4.1 Translation Overview

I/O-address translation is available as an option on interfaces configured for the IOIF (but not BIF) protocol. If I/O-address translation is enabled, the IOC supports an I/O address space of 2^{35} bytes, which is translated into a 42-bit real addresses. In this case, the IOC uses the least-significant 35 bits of the IOIF address field and ignores the remaining most-significant bits[16].

The I/O address space is divided into segments. Each segment has 2^{28} bytes, and there are 2^{7} segments. The supported I/O page sizes are 4 KB, 64 KB, 1 MB, and 16 MB. One of the four page sizes can be selected for each I/O segment. One, two, three or four of these pages sizes can be used concurrently in the system.

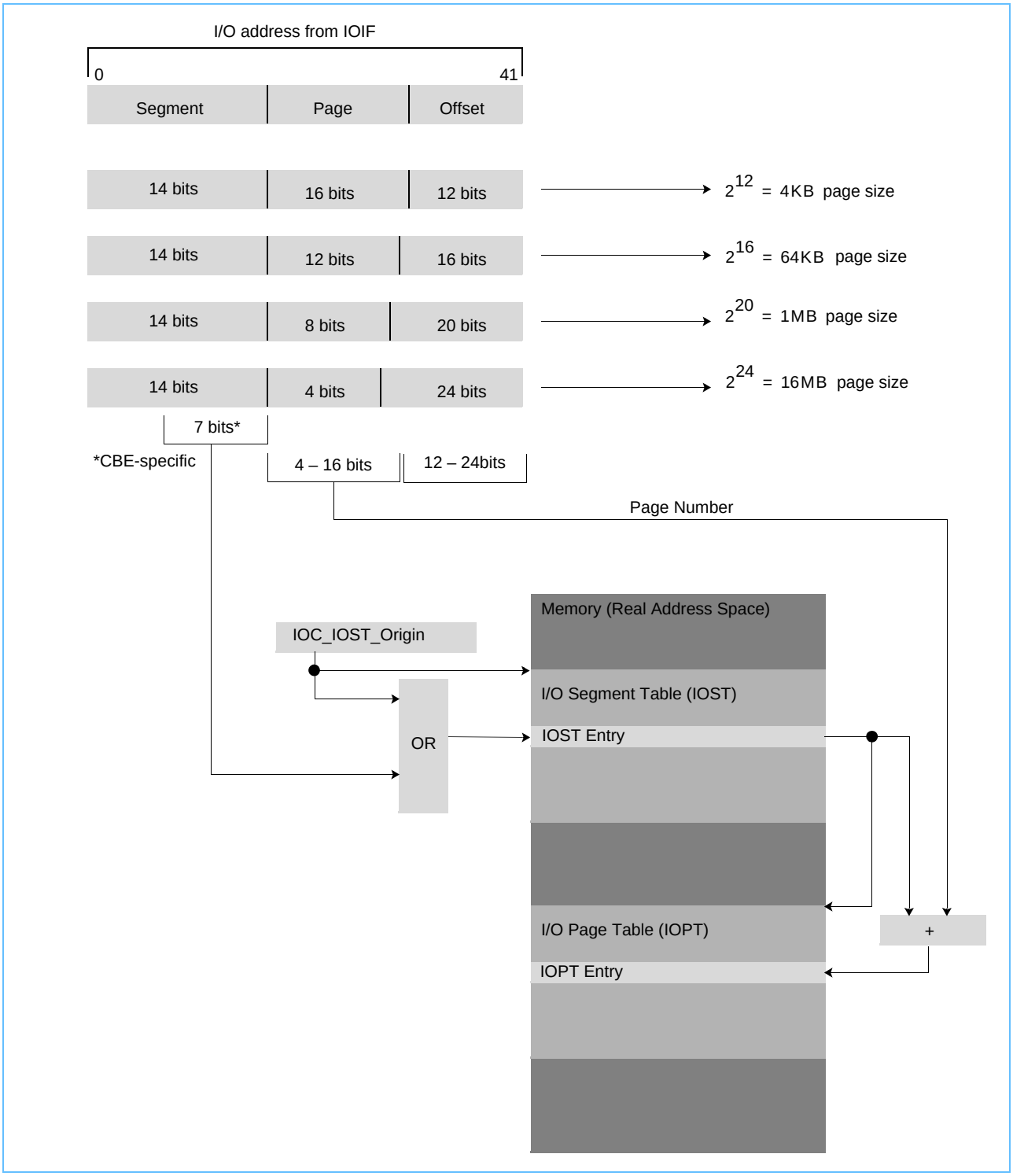
16. Bit numbering of I/O addresses in this section is based on a 42-bit I/O address with bit 0 being the most-significant and bit 41 being the least-significant.

The IOC supports an IOST cache (*Section 7.6.1* on page 182) and an IOPT cache (*Section 7.6.2* on page 184), which can be loaded by hardware or software. Multiple coherency and ordering modes support both optimized performance for I/O accesses to memory as well as coherent accesses with strong ordering when needed.

All I/O accesses by the CBE processor are big-endian accesses. If an I/O access is to a little-endian device, such as a little-endian MMIO register, software must reorder the bytes into big-endian order.

Figure 7-5 on page 178 shows an overview of the translation process in accessing the IOST and IOPT. For each of the four possible page sizes, this figure shows which bits of the I/O address are used in the CBE processor to select the IOST entry and IOPT entry. The OR operation on the `IOC_IOST_Origin` and the 7-bit segment values is equivalent to an add operation because `IOC_IOST_Origin` is naturally aligned to 4-KB boundaries.

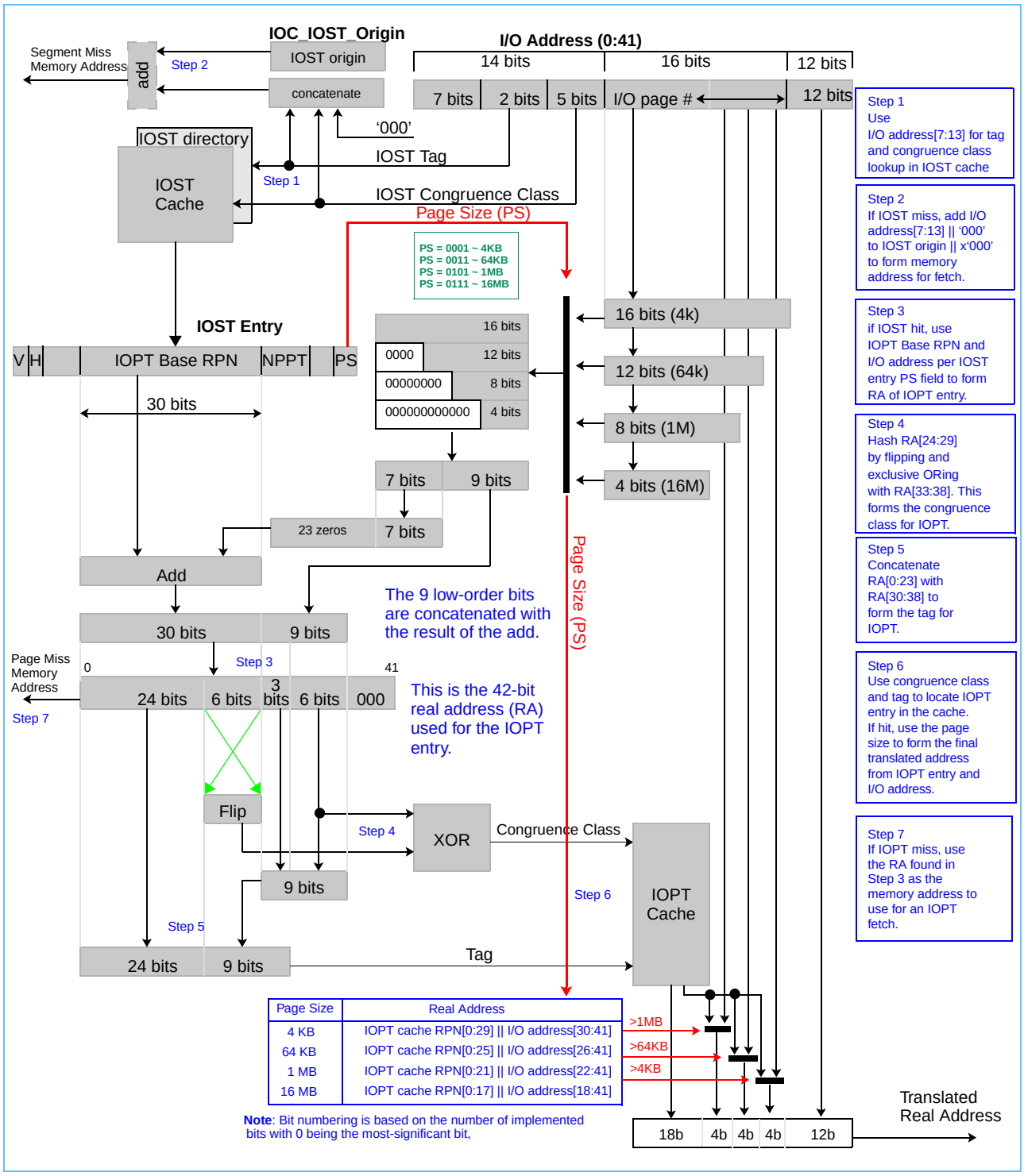
Figure 7-5. I/O-Address Translation Overview



7.4.2 Translation Steps

Figure 7-6 shows a detailed view of the I/O translation process, including details about the IOST cache and IOPT cache.

Figure 7-6. I/O-Address Translation Using Cache



I/O-address translation is enabled if `IOC_IOST_Origin[E]` = ‘1’ and `IOC_IOCmd_Cfg[TE]` = ‘1’. I/O-address translation is disabled if `IOC_IOST_Origin[E]` = ‘0’ and `IOC_IOCmd_Cfg[TE]` = ‘0’. In either case, the `IOC_IOCmd_Cfg[TE]` bit must be set to a value consistent with `IOC_IOST_Origin[E]`.

If I/O-address translation is disabled, the I/O address from the IOIF is treated as a real address. In this case, the IOC uses the least-significant 42 bits of the IOIF address and ignores the most-significant 22 bits.

If I/O-address translation is enabled, the following steps are taken by the I/O address-translation mechanism if there are misses in the IOST or IOPT cache:

1. If an IOST-cache miss occurs and `IOC_IOST_Origin[HW]` = ‘0’, an I/O segment-fault type of I/O exception occurs (*Section 7.5* on page 181), and I/O-address translation stops. If `IOC_IOST_Origin[HW]` = ‘1’, the IOST entry (*Section 7.3.3* on page 169) is loaded using the I/O address, the IOST origin field in the `IOC_IOST_Origin` register, and an index based on the IOST size field in the `IOC_IOST_Origin` register. Concatenating forms the real address of the IOST entry, as follows:
   - The most-significant (50 - S) bits of the IOST origin, where S is equal to the maximum of 0 and (IOST size - 9).
   - n ‘0’ bits, where n is equal to the maximum of 0 and (9 - IOST size).
   - t least-significant bits of the I/O address-segment bits, where t equals the value of the IOST size field.
   - Three ‘0’ bits.
2. If the IOST-entry valid bit is ‘0’, an I/O segment-fault type of I/O exception occurs and I/O-address translation stops. If the IOST-entry-valid bit is ‘1’, the I/O translation process continues with the following step.
3. The I/O address is checked to ensure that it does not access a page beyond the number of pages in the segment as defined by the NPPT field of the IOST. A page size of 2^p is specified in the IOST page-size field. For Format 1 (the only format supported in the first implementation of the *CBEA* (the CBE processor), if the value of the most-significant (28 - p) bits of the 28 least-significant I/O address bits is greater than 512 times the NPPT, an I/O page-fault type of I/O exception occurs and I/O-address translation stops. If an IOPT-cache miss occurs and `IOC_IOST_Origin[HW]` = ‘0’, an I/O page-fault type of I/O exception occurs and the I/O-address translation stops.
4. A real address is formed and used to access the IOPT entry (*Section 7.3.4* on page 171). The real address is formed by adding the following two values:
   - The IOPT-base RPN concatenated with x‘000’.
   - The most-significant (28 - p) bits of the 28 least-significant I/O address bits concatenated with ‘000’, where 2^p is the page size specified in the IOST page-size field.
5. If any of the following apply, an I/O page-fault type of I/O exception occurs and I/O-address translation stops:
   - The IOPT PP bits are ‘0’.
   - The IOPT PP bits are ‘01’ (read only), and the I/O access was a write operation.
   - The IOPT PP bits are ‘10’ (write only), and the I/O access was a read operation.
   - The IOID does not match the I/O device.

6. If the access is permitted by the IOPT PP bits, a real address is formed to use for the I/O access. For IOPT format 1, the real address is formed by concatenating bits 0 through (58 - *p*) of the IOPT RPN with the least-significant *p* bits of the I/O address. The page size specified in the IOST is 2^p.

7.5 I/O Exceptions

7.5.1 I/O Exception Causes

If I/O-address translation is enabled (`IOC_IOST_Origin[E]` = ‘1’ and `IOC_IOCmd_Cfg[TE]` = ‘1’), an I/O exception (*Section 9.6.4.4 I/O Address Translation* on page 271) can occur due to one of the following events:

- An I/O segment fault occurs in the following cases when a translation is attempted for an I/O address:
  - `IOC_IOST_Origin[HW]` = ‘0’, and an IOST cache miss occurs.
  - `IOC_IOST_Origin[HW]` = ‘1’, an IOST cache miss occurs, and there is no corresponding entry in the IOST.
- An I/O page fault occurs in the following cases when a translation is attempted for an I/O address and there is no I/O segment fault:
  - `IOC_IOST_Origin[HW]` = ‘0’ and an IOPT cache miss occurs.
  - `IOC_IOST_Origin[HW]` = ‘1’, an IOPT cache miss occurs, and there is no corresponding entry in the IOPT.
  - `IOC_IOST_Origin[HW]` = ‘1’, `IOC_IOST_Origin[HL]` = ‘1’, an IOPT cache miss occurs and all 4 entries for the IOPT cache congruence class have the H bit equal to ‘1’.
- An I/O access that is either translated or not translated and that accesses an invalid real address.
- A CBE processor access to an invalid location in the range of addresses mapped to an IOIF bus.

An I/O exception results in an external interrupt (*Section 9.6 Direct External Interrupts* on page 262). I/O exceptions caused by I/O accesses are reported to the I/O subsystem that attempted the I/O-address translation. If any of the above events are reported as an I/O exception, system error interrupt, or machine-check interrupt, and if the access is an SPE DMA get or a PPE load, the data returned contains all ones. If reported as an I/O exception, system error interrupt, or machine-check interrupt, and if the access is an SPE DMA put or a PPE store, the data is discarded. When an I/O exception occurs on an I/O access, additional information is captured in the I/O Exception Status Register (`IOC_IO_ExcpStat`); see *Section 7.5.2*, immediately below.

Note: *Software should avoid I/O exceptions. If I/O exceptions occur, they should only be a result of a software bug or a result of hardware associated with an IOIF that is not working. Whether software can or should continue operation after an I/O operation causes an I/O exception is up to the programmer or the system designer.*

7.5.2 I/O Exception Status Register

The I/O Exception Status Register (`IOC_IO_ExcpStat`) captures error information for an I/O exception. The bit fields and values for this register are described in the *Cell Broadband Engine Registers* specification.

An *IOIF device* is a device that is directly connected to a CBE IOIF port. When an I/O exception occurs on an I/O access, an Error response is sent to the IOIF device, and the following information related to the fault is captured in the `IOC_IO_ExcpStat` register:

- I/O address[7:29].
- Access type (read or write).
- I/O Device ID. (An *I/O device*, in this context, is a device that is indirectly connected to a CBE IOIF by an IOIF device.)

7.5.3 I/O Exception Mask Register

The `IOC_IO_ExcpMask`[1:2] bits are ANDed with associated I/O segment fault and I/O page fault conditions to set `IIC_ISR`[61], but these mask bits do not gate the setting of the `IOC_IO_ExcpStat` bits. See *Section 9 PPE Interrupts* on page 237 for information about when an external interrupt signal to the PPE is asserted based on these bits.

7.5.4 I/O-Exception Response

Unlike PowerPC page tables, software should normally not expect to set up an IOST or IOPT entry after an I/O exception. There is no hardware support to allow an IOIF command to be suspended at the point of an I/O exception and later restarted.

For many types of I/O operations, overruns and underruns occur if there is significant delay associated with an I/O exception. Also, the I/O reads and writes to one page may be delayed by the I/O exception while subsequent I/O reads, writes, and interrupt operations may complete, thereby violating any applicable ordering requirements and potentially giving software an erroneous indication that some I/O operation has successfully completed.

7.6 I/O Address-Translation Caches

In order to improve performance, the IOC supports an IOST cache and an IOPT cache, shown in *Figure 7-6* on page 179. These caches can be loaded by hardware or software. Both software and hardware can control the contents of these caches as described below.

7.6.1 IOST Cache

The IOST cache is direct-mapped and has 32 entries, indexed by I/O address bits [9:13]. These address bits correspond to the least-significant five bits of the I/O segment number. If `IOC_IOST_Origin[HW]` = ‘1’, a valid IOST cache entry for the I/O address does not exist, and a valid IOST entry for the I/O address exists, the IOST cache is loaded automatically by hardware. If `IOC_IOST_Origin[HW]` = ‘0’ and a valid IOST cache entry for the I/O address does not exist, an I/O segment fault occurs.

7.6.1.1 *IOST Cache Access by Software*

A doubleword load or store can be used to access each IOST cache entry in the MMIO memory map. For information about this map, see the *Cell Broadband Engine Registers* specification, which includes the offset range at which the IOST cache is accessible in the IOC address-translation MMIO memory map. For a specific 42-bit I/O address, the corresponding IOST cache-entry offset in this range is formed by using I/O address[9:13] concatenated with ‘000’.

The fields in the IOST cache entry have the same meaning as the corresponding fields in the IOST, except for the following differences:

- The H bit is not implemented and is treated as a ‘0’.
- Only the least-significant 30 bits of the IOPT Base RPN are implemented.
- Only the least-significant 3 bits of the PS field are implemented.
- There is an additional valid bit and tag field.

The IOST cache-entry valid bit and tag field correspond to values that are typically stored in a cache directory. The IOST cache-entry valid bit indicates that the IOST cache entry is valid, whereas an IOST valid bit indicates that the IOST entry is valid. There is no tag field in an IOST entry. For a given 42-bit I/O address with a corresponding valid entry in the IOST cache, the IOST cache-entry tag field has the same value as I/O address[7:8]. The valid bit should be set to ‘1’ to allow I/O-address translation to use the entry for translating I/O accesses.

After a store to the IOST cache, to guarantee that the new value of the IOST cache entry is used on future I/O-address translations, software should load from any IOST cache entry and ensure the load has completed before a subsequent store that might cause the new value to be used. See *Section 7.6.1.3 Simultaneous Hardware and Software Reload of IOST Cache* on page 184 for other considerations when `IOC_IOST_Origin[HW]` = ‘1’.

Software should ensure that, for IOST cache entries that software reloads, it does not attempt to have two or more valid entries in the IOST with the same value in the least-significant 5 bits of the I/O segment number, since the IOST cache is direct-mapped and has only 32 entries indexed by I/O address[9:13].

If software fails to pre-load an IOST cache entry, and either the hardware IOST cache-miss handling is disabled (`IOC_IOST_Origin[HW]` = ‘0’) or the IOST table does not contain the corresponding entry, an I/O segment fault occurs when an I/O device attempts to read or write the location with the corresponding the missing entry.

7.6.1.2 *IOST Cache Invalidation by Software*

An MMIO store to the IOST Cache Invalidate Register (`IOC_IOST_CacheInvd`) invalidates one entry in the IOST cache. The I/O segment to be invalidated is defined by `IOC_IOST_CacheInvd`[29:35]. Since the IOST cache is direct-mapped, the least-significant 5 bits of `IOC_IOST_CacheInvd`[29:35] specify the index of the IOST cache entry. Software must set `IOC_IOST_CacheInvd`[B] to ‘1’ when storing to the IOST Cache Invalidate Register; otherwise, the result of the store is undefined.

When the specified I/O segment has been invalidated in the IOST cache, hardware sets `IOC_IOST_CacheInvd`[B] to ‘0’. Software must only store to the IOST Cache Invalidate Register when `IOC_IOST_CacheInvd`[B] is ‘0’; otherwise, the results of the previous store and current store

to the IOST Cache Invalidate register are undefined. When `IOC_IOST_CacheInvd`[B] is set to ‘0’ by the IOC, this guarantees that I/O accesses no longer use the old cached IOST value. However, it does not guarantee that all previous I/O accesses that used the old cached IOST value have been performed. Typically, software can depend on other facilities to know that these I/O accesses have been performed—for example, notification by the I/O device by means of an external interrupt.

Software must ensure that the IOST cache invalidation has completed (`IOC_IOST_CacheInvd`[B] = ‘0’) before writing to the `IOC_IOST_CacheInvd` register, IOPT cache, `IOC_IOPT_CacheDir` register, IOST cache, or IOST cache directory. Software must also ensure that a store to the IOPT cache, `IOC_IOPT_CacheDir`, IOST cache, or IOST cache directory completes prior to storing to `IOC_IOST_CacheInvd` by executing a **sync** instruction between the last store to one of these arrays and the store to `IOC_IOST_CacheInvd`.

When `IOC_IOST_Origin[HW]` = ‘1’, software typically modifies IOST entries before invalidating the IOST cache; otherwise, the I/O-address translation facility would reload the old values if needed by an I/O access. To ensure the last store to the IOST is performed before the store to `IOC_IOST_CacheInvd`, software should use PowerPC memory-barrier instructions, such as **sync**, or should write the IOST using the storage attributes of both caching-inhibited and guarded.

Note: *Due to the asynchronous nature of DMA transfers, it is important to ensure that DMA transfers have completed before removing an IOST- or IOPT-cache invalidation. Otherwise, an I/O exception may occur.*

7.6.1.3 *Simultaneous Hardware and Software Reload of IOST Cache*

If `IOC_IOST_Origin[HW]` and `IOC_IOST_Origin[HL]` are both '1', both software and the I/O address-translation hardware can load the IOST cache when a miss occurs. This may result in hardware and software writing the same IOST cache entries.

To prevent hardware from overwriting an entry written by software, software can ensure that, for any IOST cache entry that software reloads, there are no valid entries in the IOST with the same cache congruence class. If there ever were any such entries since boot, and if software wishes to start using the IOST entry with the same congruence class, software must first make these entries in the IOST invalid, execute an enforce in-order execution of I/O (**eieio**) instruction or a synchronize (**sync**) instruction, invalidate the IOST cache, and wait for invalidation to complete before writing the IOST entry.

Software can modify an IOST cache entry when there are no outstanding I/O accesses. How software ensures that there are no outstanding I/O accesses is application-dependent.

7.6.2 IOPT Cache

The IOPT cache is 4-way set-associative and has 256 entries, indexed by a 6-bit hash formed from the I/O address, as shown in *Figure 7-7* on page 186. For each index (congruence class), a pseudo-least-recently-used (LRU) algorithm is used to determine which entry is to be replaced when hardware cache-miss handling is enabled (that is, when `IOC_IOST_Origin[HW]` = ‘1’). If `IOC_IOST_Origin[HL]` = ‘0’ and any entry in the congruence class has its hint (H) bit or valid (V) bit set to ‘0’, this pseudo-LRU algorithm avoids replacing a valid entry that has its H bit set to ‘1’. If `IOC_IOST_Origin[HL]` = ‘1’, this pseudo LRU algorithm does not replace an entry that has its H bit set to ‘1’, even if the valid bit for the IOPT cache entry is ‘0’.

7.6.2.1 *IOPT Cache Access by Software*

Doubleword loads or stores can be used to access the IOPT cache directory (`IOC_IOPT_CacheDir`) register in the MMIO memory map. A load of `IOC_IOPT_CacheDir` simultaneously causes the corresponding IOPT cache entry to be read into the IOPT Cache (`IOC_IOPT_Cache`) register. Similarly, a store to `IOC_IOPT_CacheDir` simultaneously causes the corresponding IOPT cache entry to be written from `IOC_IOPT_Cache`.

As shown in the *Cell Broadband Engine Registers* specification, `IOC_IOPT_CacheDir` register is accessible at a set of offset ranges in the IOC address-translation MMIO memory map. Each way of `IOC_IOPT_CacheDir` is accessed using a different 512-byte range. For a specific 42-bit I/O address, the corresponding `IOC_IOPT_CacheDir` entry offset in this range is formed by adding of two quantities:

- x'0200' times 1 minus the number of the way of `IOC_IOPT_CacheDir`.
- 6 bits of the IOPT cache hash concatenated with '000'.

Software can read an entry in the IOPT cache by:

1. Performing a doubleword load from the `IOC_IOPT_CacheDir` entry. This simultaneously causes the corresponding IOPT cache entry to be read into `IOC_IOPT_Cache`.
2. Performing a doubleword load from `IOC_IOPT_Cache`.

Software can write an entry in the IOPT cache by:

1. Storing a doubleword to `IOC_IOPT_Cache`. This store does not cause an IOPT cache entry to be updated.
2. Storing a doubleword to the `IOC_IOPT_CacheDir` entry. This store causes `IOC_IOPT_CacheDir` to be loaded with the store data and causes the IOPT cache to be simultaneously loaded with the value in `IOC_IOPT_Cache`. Hardware-generated parity bits are also automatically loaded. The tag, as determined in *Section 7.6.2.2 IOPT Cache Hash and IOPT Cache Directory Tag* on page 186, should be written into the tag field of `IOC_IOPT_CacheDir` entry and the valid bit should be set to '1' to allow I/O-address translation to use this new entry for translating I/O accesses.

Note: *Software must not store to the IOPT Cache Invalidate (`IOC_IOPT_CacheInvd`) register between the preceding steps 1 and 2 of the read or write sequence; otherwise, the result of step 2 is undefined.*

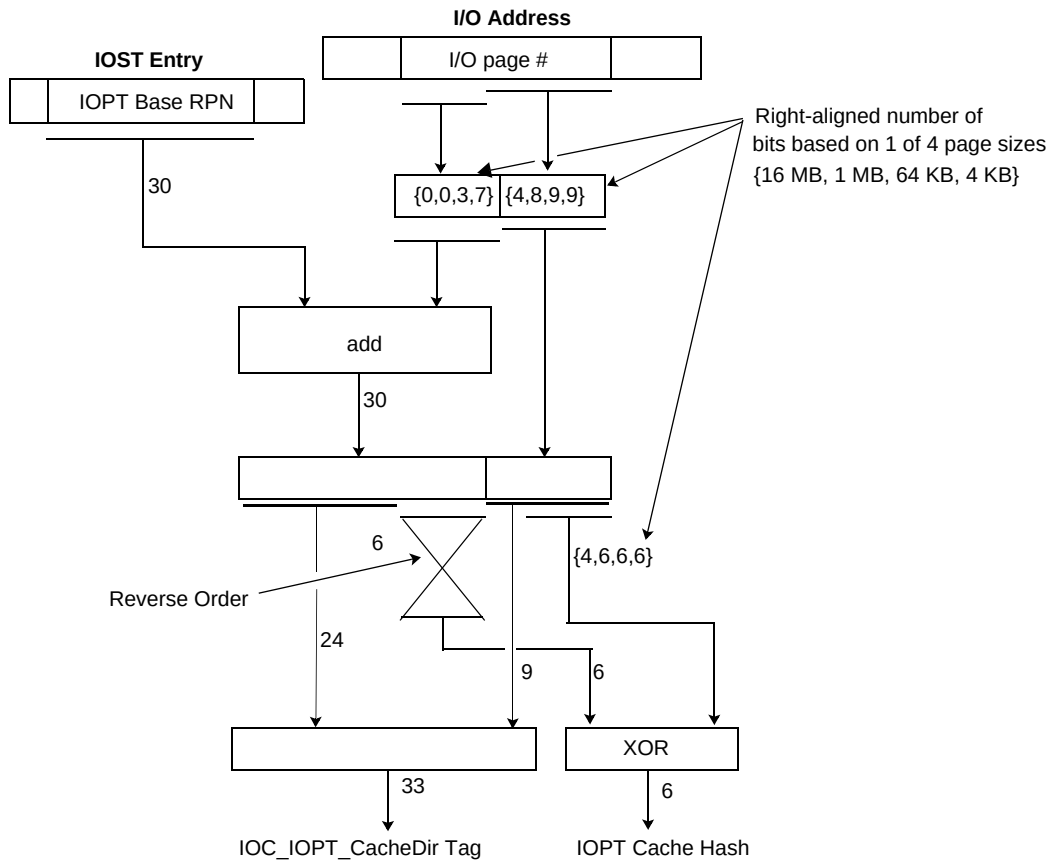
After a store to `IOC_IOPT_CacheDir`, to guarantee that the new value of the IOPT cache entry is used on future I/O-address translations, software should load from any `IOC_IOPT_CacheDir` entry and ensure the load has completed before a subsequent store that might cause the new value to be used. See *Section 7.6.2.4 Simultaneous Hardware and Software Reload of IOPT Cache* on page 188 for other considerations when `IOC_IOST_Origin[HW]` = '1'.

Software must not write two or more ways of the `IOC_IOPT_CacheDir` for the same congruence class with valid entries having the same tag value; otherwise, the results of subsequent I/O-address translations are undefined.

7.6.2.2 *IOPT Cache Hash and IOPT Cache Directory Tag*

If hardware I/O-address translation is enabled, software must know how the IOPT cache hash is created in order to avoid setting too many bits in the same congruence class. The 6-bit IOPT cache hash is used by the hardware to select the IOPT cache congruence class for hardware reloads. The generation of this hash value is shown in *Figure 7-7* on page 186.

Figure 7-7. IOPT Cache Hash and IOPT Cache Directory Tag



The generation of the hash value in *Figure 7-7* is based on the following steps:

1. Select 4, 8, 9, or 9 bits of the page number from the I/O address, based on the page size of 16 MB, 1 MB, 64 KB, or 4 KB, respectively. The page size is defined by the IOST entry PS field.
2. Select the most-significant 0, 0, 3, or 7 bits of the page number from the I/O address, based on the page size of 16 MB, 1 MB, 64 KB, or 4 KB, respectively.
3. Add the value selected in step 2 to the 30-bit IOPT Base RPN from the IOST entry.
4. Select the least-significant 6 bits of the result from the previous step.
5. Put these bits in reverse order; that is, swap bits 0 and 5, swap bits 1 and 4, and swap bits 2 and 3.

6. From the bits selected in step 1, select the least-significant 4, 6, 6, or 6 bits, based on the page size of 16 MB, 1 MB, 64 KB, or 4 KB, respectively.
7. If the page size is 16 MB, create a 6-bit value from two ‘0’ bits concatenated with the 4 bits from the previous step; otherwise, create a 6-bit value equivalent to the value created in the previous step.
8. Exclusive-OR the 6 bit value from step 5 with the 6-bit value from the previous step.

The result of the exclusive-OR in step 8 is the IOPT cache hash.

The `IOC_IOPT_CacheDir` tag is determined by the following additional steps:

a. Create a 9-bit value by padding as many ‘0’ bits as necessary to the left of the bits from step 1 above.
b. Form a 33-bit `IOC_IOPT_CacheDir` tag by concatenating the most-significant 24 bits from the addition result in step 3 with the 9-bit value from the previous step.

7.6.2.3 *IOPT Cache Invalidation by Software*

An MMIO store to `IOC_IOPT_CacheInvd` invalidates IOPT cache entries corresponding to a number of IOPT entries. The value in the NE field in `IOC_IOPT_CacheInvd` is equal to one less than the number of IOPT entries to be invalidated in the IOPT cache. The MMIO store invalidates all IOPT cache entries corresponding to IOPT entries located at the doubleword addresses between (IOPTE_DRA || ‘000’) and ((IOPTE_DRA + NE) || ‘000’), inclusive. The IOPTE_DRA field in `IOC_IOPT_CacheInvd` specifies the I/O Page Table Entry Doubleword Real Address, where (IOPTE_DRA || ‘000’) is the real address of the first IOPT entry to be invalidated. Software must set `IOC_IOPT_CacheInvd`[B] to ‘1’ when storing to `IOC_IOPT_CacheInvd`; otherwise, the result of the store is undefined.

When the IOPT cache entries have been invalidated for the specified IOPT entries, hardware sets the busy bit, `IOC_IOPT_CacheInvd`[B], to ‘0’. Software must only store to the IOPT cache invalidate register when `IOC_IOPT_CacheInvd`[B] is ‘0’; otherwise, the results of the previous store and current store to `IOC_IOPT_CacheInvd` are undefined. When `IOC_IOPT_CacheInvd`[B] is set to ‘0’ by the IOC, this guarantees that I/O accesses no longer use the old cached IOPT value. However, it does not guarantee that all previous I/O accesses that used the old cached IOPT value have been performed. Typically, software can depend on other facilities to know that these I/O accesses have been performed—for example, notification by the I/O device by means of an external interrupt.

An invalidation of an IOPT cache entry does not change the value of the H bit in the cache.

If the I/O-address translation facility already has an outstanding load of an IOPT entry when software does an IOPT cache invalidation that includes that IOPT entry, then hardware will not set `IOC_IOPT_CacheInvd`[B] to ‘0’ until the outstanding load completes and either the reloaded IOPT cache entry is invalidated or the reload is somehow aborted.

Software must ensure that the IOPT cache invalidation has completed (`IOC_IOPT_CacheInvd`[B] = ‘0’) before writing `IOC_IOST_CacheInvd`, IOPT cache, `IOC_IOPT_CacheDir`, IOST cache, or IOST cache directory. Software must ensure that a store to the IOPT cache, `IOC_IOPT_CacheDir`, IOST cache or IOST cache directory completes prior to storing to `IOC_IOPT_CacheInvd` by executing a **sync** instruction between the last store to one of these arrays and the store to `IOC_IOPT_CacheInvd`.

When `IOC_IOST_Origin[HW]` = ‘1’, software typically modifies IOPT entries before invalidating the IOPT cache; otherwise, the I/O-address translation facility would reload the old values if needed by an I/O access. Software should use PowerPC memory-barrier instructions, such as **sync**, to ensure that the last store to the IOPT is performed before the store to `IOC_IOPT_CacheInvd` or software should write the IOPT using the storage attributes of both caching-inhibited and guarded.

7.6.2.4 *Simultaneous Hardware and Software Reload of IOPT Cache*

If `IOC_IOST_Origin[HW]` and `IOC_IOST_Origin[HL]` are both '1', both software and the I/O address-translation hardware can load the IOPT cache when a miss occurs. This may result in hardware and software writing the same IOPT cache entries.

To prevent hardware from overwriting an entry written by software, software must set the H bits in the IOPT cache to ‘1’. To prevent hardware from writing an entry that software is changing at the time, software must only modify IOPT cache entries that already have the H bit equal to ‘1’. At boot time, if both software and hardware reloads are to be used, software should reserve a number of entries in each congruence class by setting their H bits to ‘1’ before setting `IOC_IOST_Origin[HW]` to ‘1’.

Software should only modify an IOPT cache entry for which the H bit is a ‘0’ when there are no outstanding I/O accesses. How software ensures that there are no outstanding I/O accesses is application-dependent. If there is an outstanding IOPT cache miss at the time software writes the H = ‘0’ entry, hardware cache-miss handling may overwrite the entry that software modified even though software set the H bit to ‘1’.

Alternatively, software may use the techniques described in *Section 7.6.1.3* on page 184 for IOST cache loading.

Invalidation of an IOPT cache entry will not change the value of the H bit in the cache. This allows software to use the hardware-invalidation mechanisms and still retain the ability to later load the entries that were reserved for software use (that is, entries with H = ‘1’).

Software may also preload the IOPT cache even when `IOC_IOST_Origin[HL]` = ‘0’. However, if `IOC_IOST_Origin[HW]` = ‘1’ and there is a simultaneous I/O access that has an IOPT cache miss in the same congruence class, then hardware miss handling might cause the software entry to be overwritten. If an IOPT cache miss occurred just before software wrote the entry, hardware miss handling might have already selected the entry to be written when the IOPT data arrives from memory. At the time hardware detects an IOPT cache miss, hardware selects an entry in the congruence class, using the first entry to meet certain criteria according to the following order and criteria:

1. An entry with the valid bit equal to ‘0’.
2. An entry with the H bit equal to ‘0’, using the pseudo-LRU algorithm among such entries.
3. An entry with the H bit equal to ‘1’, using the pseudo-LRU algorithm among such entries.

If hardware selects an entry that software subsequently writes before hardware updates the entry from the IOPT, the entry that software writes is overwritten when the IOPT data arrives from memory.

7.7 I/O Storage Model

The *PowerPC Architecture* and the *Cell Broadband Engine Architecture* (CBEA) define the order of storage operations by the PPE and SPEs within the CBE coherence domain (all CBE processor elements and all interfaces to main storage). These documents also define the order of storage operations by the PPE and SPEs relative to I/O mechanisms. This section clarifies the order of storage operations by the PPE and SPEs relative to an I/O interface (IOIF), because ordering operations defined by the *PowerPC Architecture* (such as, **sync**) have no corresponding equivalent on the IOIFs.

This section also describes the order of operations on an IOIF and its relationship to the order in which the operations are performed within the CBE coherence domain or on another IOIF. Accesses to locations outside the CBE coherence domain are considered by the coherence domain to be performed when they are considered performed on the IOIF. From the perspective of the CBEA, these accesses are performed coherently in the coherence domain of the I/O subsystem in the same order as they are performed on the IOIF. However, an I/O subsystem may have a different perspective. It is the responsibility of the I/O subsystem that attaches to the IOIF to define ordering rules for the completion of IOIF operations in the coherence domain of the I/O subsystem.

CBE-processor systems map I/O units to main storage. Storage operations that cross the CBE coherence-domain boundary are referred to as *I/O operations*. These accesses can be initiated either internally in the CBE processor (called *outbound* accesses) or externally (called *inbound* accesses). An access initiated externally is called an *I/O access*, and the units that initiate them are called *I/O units*. An I/O unit can be one or more physical I/O device(s), I/O bridge(s), or other functional unit(s) attached to an IOIF, in which one value of the I/O Identifier (IOID) described in the IOPT is used for all accesses by these physical entities.

A physical IOIF may be divided into several virtual IOIFs called *virtual channels*. An access on one virtual channel must be allowed to pass accesses for other virtual channels so that a slower virtual channel does not block a faster virtual channel. (See the *Hardware Initialization Guide* for a description of I/O virtual channels.)

7.7.1 Memory Coherence

I/O accesses—inbound accesses from an I/O unit to the CBE processor's Element Interconnect Bus (EIB)—can be treated as either memory-coherence-required or not, depending on the following control bits:

- *IOC TE Bit*—This is the enable bit for I/O-address translation. It is located in the IOCmd Configuration Register (`IOC_IOCmd_Cfg`). Translation is enabled when TE = '1'. See the *Cell Broadband Engine Registers* specification for details.
- *IOIF C Bit*—This is the memory-coherence bit sent with each command on the IOIF. Memory coherence is required on any read or write when C = '1', and memory coherence is not required when C = '0'.
- *IOPT M Bit*—This is the memory-coherence bit from the IOPT entry or IOPT-cache entry corresponding to the accessed location, as described in *Section 7.3.4* on page 171. Memory coherence is required when M = '1', and memory coherence is not required when M = '0'.

Table 7-2 summarizes the effects on memory coherence of I/O accesses for the various combinations of control bits. If either C = ‘0’ or M = ‘0’, the access is performed on the EIB as not coherent. For incoming I/O addresses that are not translated, the IOIF C bit is used to determine coherence requirements. If this bit is ‘0’, the access is performed on the EIB as not coherent. The access is also performed on the EIB as not coherent if the address is in the range defined by the EIB Local Base Address Register 0 (`EIB_LBAR0`) and the EIB Local Base Address Mask Register 0 (`EIB_LBAMR0`)—or by the `EIB_LBAR1` and `EIB_LBAMR1` registers.

Table 7-2. Memory Coherence Requirements for I/O Accesses

Control Bit Combination[1]			Memory-Coherence-Required Attribute Used for the I/O Access
TE	C	M	
x	0	x	Not coherent
0	1	x	Memory-coherence-required
1	1	0	Not coherent
1	1	1	Memory-coherence-required

1. An “x” indicates don’t care or not applicable.

There is no hardware support for maintaining I/O-subsystem memory coherence relative to caches in the CBE processor. Any data in the I/O subsystem must be accessed with the storage attribute of caching-inhibited, or software must manage its coherency. For storage that does not have the memory-coherence-required attribute, software must manage memory coherence to the extent required for the program to run correctly. The operations required to do this may be system-dependent. Because the memory-coherence-required attribute for a given storage location is of little use unless the PPE, all SPEs, and all I/O units that access the location do so coherently, statements about memory-coherence-required storage elsewhere in this section generally assume that the storage has the memory-coherence-required attribute for all the PPE, SPEs, and I/O units that access it.

7.7.2 Storage-Access Ordering

The following sections describe both outbound and inbound ordering of storage accesses between the EIB and the IOIFs. For background about CBE storage ordering, see *Section 20.1 Shared-Storage Ordering* on page 551.

7.7.2.1 *Outbound (EIB to IOIF) Accesses*

All outbound read and write accesses to I/O locations mapped to main storage locations are performed in program order. The IOIF C bit (*Section 7.7.1* on page 189) and the IOIF S bit (*Section 7.7.2.2* on page 191) are driven to indicate memory-coherence-required on all accesses by the PPE or SPEs. Outbound write data associated with write accesses is sent on the IOIF in the order in which acknowledgement responses are received from the I/O unit on the IOIF.

To be more specific, CBE-processor outbound accesses are performed on the IOIF in a weakly consistent storage order, except as ordered by one of the following rules:

- Accesses performed by a PPE or SPE to the same location beyond the IOIF are performed in a strongly ordered manner on the IOIF. (The “same location” is defined as any byte locations that match.) Thus, the same locations may be accessed even if the initial addresses used for the accesses do not match.

- If a PPE performs two stores to different locations across the same IOIF, and those storage locations are both caching-inhibited and guarded, the stores are performed on the IOIF in program order.
- If an SPE performs two writes to different locations across the same IOIF, if the locations are caching-inhibited, and if the writes are ordered by a **fence** or **barrier**, the writes are performed on the IOIF in program order. The SPE writes can be DMA transfers or send-signal instructions with fence or barrier options (**sndsig<*f,b*>**). See the *Cell Broadband Engine Architecture* for details on which writes are ordered by a **fence** or **barrier**, details on tag-specific ordering, and details on SPE read ordering. Since reads are necessarily completed on the IOIF before they can be completed in a CBE processor, *Cell Broadband Engine Architecture* ordering rules are sufficient to establish read order on the IOIF.
- When a PPE executes a **sync** or **eieio** instruction, a memory barrier is created which orders applicable storage accesses pairwise as defined by the *PowerPC Architecture*. The ordering defined by the *PowerPC Architecture* applies to the order in which the applicable storage accesses are performed on the IOIF.
- When an SPE executes an MFC synchronize (**mfcsync**) or MFC enforce in-order execution of I/O (**mfceieio**) command, a memory barrier is created for DMA transfers within the same DMA tag group[17]. This orders applicable storage accesses pairwise as defined by the *Cell Broadband Engine Architecture* and *PowerPC Architecture*. The ordering defined by these architectures applies to the order in which the applicable storage accesses are performed on the IOIF.
- If an interrupt is transmitted by a PPE or an SPE to a unit on the IOIF, previous loads, stores, or DMA transfers by that PPE or SPE to locations beyond the same IOIF are performed on the IOIF before the interrupt is transmitted on the IOIF.

In general, non-overlapping accesses caused by different PPE load instructions that specify locations in caching-inhibited storage may be combined into one access. Non-overlapping accesses caused by separate store instructions that specify locations in caching-inhibited storage may also be combined into one access. Such combining does not occur if the load or store instructions are separated by a **sync** instruction, or by an **eieio** instruction if the storage is also guarded. Such combining does not occur for SPE accesses to locations beyond the IOIF that are ordered by a **barrier**, **mfceieio**, or **mfcsync** command, or by another MFC command that has a barrier or fence option.

For ordering accesses, the IOC makes no distinction between a specific PPE or SPE. All EIB read and write commands to the IOIF are transmitted on the IOIF in the order in which they are performed on the EIB, regardless of the unit initiating the access. If **eieio** instructions are configured to be propagated to an IOIF, all **eieio**, read, and write commands from the EIB are transmitted on the IOIF in the order in which they are performed on the EIB. An I/O device can use **eieio** to determine which read and write commands need to be ordered instead of performing all read and write commands in the order in which they are received.

7.7.2.2 *Inbound (IOIF to EIB) Accesses*

The ordering of inbound storage accesses is determined by the following control bits:

17. The *Cell Broadband Engine Architecture* specifies that the **mfcsync** and **mfceieio** commands are tag-specific, but the first implementation of the *CBEA* (the CBE processor) treats all three barrier command identically, having no tag-specific effects.

- *IOC TE Bit*—See *Section 7.7.1* on page 189.
- *IOIF S Bit*—This is the strict-ordering bit sent with each command on the IOIF. Strict ordering is required on any read or write when S = ‘1’, unless the SO bits (see immediately below) override this requirement. Weak ordering is permitted when S = ‘0’.
- *IOPT SO Bits*—These are the storage-ordering bits from the IOPT entry or IOPT-cache entry corresponding to the accessed location, described in *Section 7.3.4* on page 171.

Table 7-3 summarizes the effects on inbound read and write storage order for the various combinations of control bit. The table assumes that the accesses are from one I/O unit on one virtual channel on one IOIF. Inbound data transfers from the IOIF to the EIB that are associated with these commands are kept in the order in which they are received by the EIB.

Table 7-3. Ordering of Inbound I/O Accesses from Same or Different I/O Addresses

Control Bit Combination[1]			Same IOID[2] and VC[3], Same I/O Address	Same IOID and VC, Different I/O Address, Previous Command Was a Write	Same IOID and VC, Different I/O Address, Previous Command Was Not a Write
TE	S	SO			
0	0	x	Ordered	Not Ordered	Not Ordered
0	1	x	Ordered	Ordered	Ordered
1	0	00	Ordered	Not Ordered	Not Ordered
1	0	01	Undefined (SO = ‘01’ is reserved)		
1	0	10	Ordered	Not Ordered	Not Ordered
1	0	11	Ordered	Not Ordered	Not Ordered
1	1	00	Ordered	Not Ordered	Not Ordered
1	1	01	Undefined (SO = ‘01’ is reserved)		
1	1	10	Ordered	Ordered	Not Ordered
1	1	11	Ordered	Ordered	Ordered

1. An “x” indicates don’t care or not applicable.
2. IOID = I/O Identifier.
3. VC = I/O Virtual Channel. See the *Hardware Initialization Guide* for a description of I/O virtual channels.

Table 7-4 differs from *Table 7-3* in that the accesses are (a) only from different I/O addresses, (b) to locations that are either not cached by the PPE or have the memory-coherence-required attribute, and (c) made with no assumption about whether the previous access was or was not a write.

Table 7-4. Ordering of Inbound I/O Accesses from Different I/O Addresses

Control Bit Combination[1]			Ordering of I/O Accesses[2]
TE	S	SO	
x	0	x	Previous reads and writes NOT necessarily performed before this access.
0	1	x	Previous reads and writes performed before this access.
1	1	00	Previous reads and writes NOT necessarily performed before this access.
1	1	01	Undefined (SO = ‘01’ is reserved).
1	1	10	Previous writes performed before this access.
1	1	11	Previous reads and writes performed before this access.

1. An “x” indicates don’t care or not applicable.
2. Assumes accesses are reads or writes from different I/O addresses, accesses are from one IOIF bus with the same IOID and virtual channel, and addressed locations either have the storage attribute of memory-coherence-required or are not in a PPE cache.

If either S = ‘0’ or SO = ‘00’, the access on the EIB may be performed out of order relative to a preceding access, unless the address, I/O Identifier (IOID), and I/O Virtual Channel (VC) of both accesses match. For incoming I/O addresses that are not translated, the IOIF S bit is used to determine ordering requirements.

Aside from interrupt commands, only read or write commands that have the same IOID and VC are ordered. Interrupts are ordered behind all previous writes from the same IOIF interface. Interrupt-reissue commands are intercepted before they reach the EIB, and therefore have no ordering properties.

I/O accesses to the EIB from an IOIF are ordered based on the implicit storage ordering for real addresses if the I/O address is not translated, or are ordered based on the SO bits if the I/O address is translated. This ordering can be further relaxed if S = ‘0’.

Ordering Rules

Inbound I/O accesses are unordered relative to the PPE and SPEs, except in the following cases; the SO value shown in these exceptions is either the implicit SO for real addresses, if the I/O address is not translated, or the IOPT SO bits if the I/O address is translated:

- Accesses by a specific I/O unit on one IOIF virtual channel to the same location in the CBE processor, are performed (with respect to the PPE and SPEs) in the order in which they are initiated on the IOIF if the location has the storage attribute of memory-coherence-required or if the location is not in a cache in the CBE processor. The address is used to determine whether the “same location” that is accessed is the I/O address passed on the IOIF. (The “same location” is defined as any byte locations that match.)

  Note: *If an I/O device reads from a CBE memory location using one I/O address, and an I/O device writes to the same CBE memory location using a different address, these operations need not be kept in order. To have these operations performed in the expected I/O order, the system architect must use the same virtual channel and the same IOID on the IOIF for all I/O devices on an I/O bus with a shared-memory address space, such as PCI Express.*

- An I/O unit write, with corresponding SO = ‘10’ and S = ‘1’, will be performed with respect to the PPE and SPEs after all previous writes by the same I/O unit on the same IOIF virtual channel have been performed with respect to the PPE and SPEs. This is true only for previous writes to locations that have the storage attribute of memory-coherence-required, or that are not in a PPE cache. This is true regardless of the SO values of the previous writes.
- An I/O unit read or write, with a corresponding SO = ‘11’ and S = ‘1’, will be performed with respect to the PPE and SPEs after all previous reads and writes on the same IOIF virtual channel by the same I/O unit. This is true only for previous accesses to locations that have the storage attribute of memory-coherence-required or that are not in a PPE cache. This is true regardless of the SO values of the previous accesses.
- Data from an I/O read must be allowed to pass on the IOIF prior to a CBE-processor access that has been previously queued in the IOC, in order to prevent a possible deadlock. For example, read data from an SPE’s LS must be allowed to pass to the IOIF before a CBE write command that is queued in the IOC waiting for a command credit.

- Data from an I/O write must be allowed to pass on the IOIF to CBE memory before another I/O access to the same coherency block is performed on the internal CBE interconnect bus, in order to prevent a possible deadlock. For example, an I/O device might send two write commands to the SPU Signal Notification Register 1. Until data for the first write is received, the second write may never get a response on the IOIF. After the SPE receives the data for the first write, the second write command would get acknowledged on the IOIF. Thus data for the first write must be sent without waiting for a response for the second write command. A *coherency block* is a collection of memory bytes corresponding to a cache line.
- Typically, software is notified of the completion of an I/O operation by means of an interrupt transmitted on the IOIF after the last I/O read or write required by the I/O operation.
  - Software must ensure that, after the interrupt, software sees the effect of the previous I/O writes performed by the I/O unit assuming that all writes and the interrupt are transmitted on the same IOIF virtual channel. If the I/O subsystem does not ensure that previous I/O writes have occurred on the IOIF before a subsequent interrupt is transmitted on the IOIF, additional system- or I/O-subsystem-specific methods may be required to ensure that software sees the effect of previous I/O writes.
  - The I/O unit must not transmit an interrupt on the IOIF until the last I/O read associated with the interrupt has been performed on the IOIF.

The ordering rules above do not apply to an access that causes an I/O exception. Other preceding and subsequent accesses are still ordered according to these rules.

Read or write accesses from an IOIF to the CBE processor are never combined into one access.

Cache Writebacks

Before an I/O write is performed to a storage location that does not have the memory-coherence-required attribute, software must ensure that modified versions of the memory location in any PPE cache or SPE atomic cache[18] are written to memory by using PPE cache-management instructions and/or SPE synchronization instructions or MFC commands. Otherwise, if the cached copy is modified, the modified line could subsequently be written to memory, overlaying the memory copy written by the I/O device. After the modified cache copies are written to memory, software must ensure that a cached copy does not become modified until after the I/O write is performed. If deemed necessary, privileged software could use the following methods to ensure this:

- Software could assign the location the storage attribute of caching-inhibited.
- Software could remove the corresponding valid entry from the page table and TLBs.
- Software could use the page protection bits to prevent writes to the location.

See the *PowerPC Architecture* for details about caching-inhibited, page table, TLB, and page protection bits.

18. Four cache lines are available in each SPE for atomic operations.

Interrupts

Regardless of the virtual channel and IOID, interrupt commands from an IOIF are not performed on the EIB until all previous writes from the IOIF are successfully performed on the EIB. “Successfully performed” means the writes have obtained a combined response of acknowledged and not retry. This does not guarantee the data transfer for a previous write has completed. However, if the PPE accesses such a location after reading the corresponding interrupt information by means of the MMIO register `IIC_IPP0` or `IIC_IPP1`, this PPE access is performed after the previous IOIF write. Thus, if this PPE access to such a location is a load, the load returns the data written by the previous IOIF write, assuming there is no intervening write to the location.

Because the data transfer for a write from the IOIF is not guaranteed to be completed when the subsequent interrupt is presented to the PPE, and because the SPEs do not operate in the EIB coherence domain, and because the interrupt does not have a cumulative ordering property, if a subsequent SPE access to its LS is a result of a subsequent PPE store, the SPE access can still occur before the data transfer for the write from the IOIF. For example, if the PPE stores to an SPU signal-notification register after receiving the interrupt, and the SPE accesses LS after the signal notification event, the SPE access could occur before the data transfer for the write from the IOIF. As described in the *Section 20.1.5* on page 567, the MFC Multisource Synchronization Facility can be used to ensure that such previous data transfers have completed.

7.7.3 I/O Accesses to Other I/O Units Through an IOIF

Note: *The description below conforms to the I/O architecture. However, unlike the I/O architecture, the first implementation of the CBEA (the CBE processor) does not support operations from an IOIF that access the same IOIF.*

An I/O access from an IOIF may access the same or a different IOIF. The IOIF on which the I/O command occurred is called the incoming IOIF, or the IOIF in bus. The IOIF that is the target of the I/O command is called the outgoing IOIF, or the IOIF out bus.

I/O accesses from an IOIF to a different IOIF are ordered based on the implicit SO for real addresses, if the I/O address is not translated, or based on the IOPT SO, if the I/O address is translated. This ordering can be further relaxed if the S bit = ‘0’. I/O accesses from an IOIF to a different IOIF are generally unordered on the outgoing IOIF, except in the following cases:

- Accesses by a specific I/O unit on one IOIF virtual channel that access the same location beyond the outgoing IOIF are performed on the outgoing IOIF in the order they are initiated on the incoming IOIF. For I/O accesses, the address used to determine whether the “same location” is accessed is the I/O address passed on the IOIF. (The “same location” is defined as any byte locations that match.)
- An I/O-unit write with a corresponding SO = ‘10’ and S = ‘1’ will be performed on the outgoing IOIF after all previous writes by the same I/O unit on the same IOIF virtual channel have been performed on the outgoing IOIF, regardless of the storage attribute of memory-coherence-required or the SO values of the previous writes.
- An I/O-unit read or write with a corresponding SO = ‘11’ and an S = ‘1’ will be performed on the outgoing IOIF after all preceding reads and writes on the same IOIF virtual channel by the same I/O unit, regardless of the storage attribute of memory-coherence-required or the SO values of the previous accesses.

- An I/O-unit interrupt will be performed on the outgoing IOIF after all previous writes by the same I/O unit on the same IOIF virtual channel have been performed on the outgoing IOIF, regardless of the storage attribute of memory-coherence-required or the SO values of the previous writes.
  - If the I/O subsystem does not ensure that previous I/O writes have occurred on the IOIF before a subsequent interrupt packet is transmitted on the IOIF, additional system or I/O subsystem-specific methods may be required to ensure that software in the destination IOIF sees the effect of previous I/O writes.
  - The I/O unit must not transmit an interrupt on the IOIF until the last I/O read associated with the interrupt has been performed on the IOIF.

Read or write accesses from an IOIF to a different IOIF are never combined into one access.

7.7.4 Examples

The examples in this section reference combinations of the following control bits:

- *IOIF C Bit*—This is the memory-coherence bit sent with a command on the IOIF. Memory coherence is required on any read or write when C = ‘1’, and memory coherence is not required when C = ‘0’.
- *TLB I Bit*—This is the caching-inhibited bit from the TLB Real Page Number Register (`TLB_RPN`), as described in the *Cell Broadband Engine Registers* specification. Caching is inhibited when I = ‘1’, and caching is permitted when I = ‘0’.
- *IOPT M Bit*—This is the memory-coherence bit from the IOPT entry or IOPT-cache entry corresponding to the accessed location, as described in *Section 7.3.4* on page 171. Memory coherence is required when M = ‘1’, and memory coherence is not required when M = ‘0’.
- *IOIF S Bit*—This is the strict-ordering bit sent with a command on the IOIF. Strict ordering is required on any read or write when S = ‘1’, unless the SO bits (see immediately below) override this requirement. Weak ordering is permitted when S = ‘0’.
- *IOPT SO Bits*—These are the storage-ordering bits from the IOPT entry or IOPT-cache entry corresponding to the accessed location, described in *Section 7.3.4* on page 171.

In the first example that follows, the ordering of an outgoing DMA transfer with respect to a subsequent send-signal is described. The subsequent examples demonstrate the order of commands originating externally on an IOIF relative to commands originating inside the CBE processor. Unless otherwise stated, locations A and B in these examples refer to memory that is in the CBE coherence domain.

7.7.4.1 *Example 1*

Consider the following example of an SPE initiating an outbound DMA transfer to an I/O device, followed by the SPE sending a signal notification to a different location on that I/O device. On all outgoing I/O accesses, the C and S bits are set to ‘1’ by hardware; in this example, the I bit in the TLB is assumed to be ‘1’:

1. DMA transfer to location A in the I/O device.
2. **sndsig <f,b>** to location B in the I/O device.

The write command to location A is performed on the IOIF before the write command to location B is performed on the IOIF. There is no guarantee that the data is transferred on the IOIF by the time that the write command to location B is performed. However, the data is transmitted on the IOIF in the order the commands are acknowledged on the IOIF. In some cases the write to location B might cause some operation to be initiated in an I/O bridge or I/O device, and this operation might have a dependency on the data for location A. In this case, the I/O bridge or I/O device must ensure the data for location A has arrived at the necessary destination point before initiating the operation triggered by the write to location B.

7.7.4.2 *Example 2*

Software must ensure that PPE thread 0 does not have a modified copy of location A in its cache when the write to location A is performed. If PPE thread 0 might have an unmodified copy of location A, then software must execute a **dcbf** instruction to the location to ensure that PPE thread 0 loads data from memory. For IOIF1 virtual channel 3, C and S are the IOIF address-modifier bits; SO and M are bits from the IOPT entries corresponding to the locations A and B. For PPE thread 0, the loads use translated addresses and use M bits from the *PowerPC Architecture* page table entry (PTE) corresponding to locations A and B. M = ‘1’ means the access has a storage attribute of memory-coherence-required.

IOIF1 Virtual Channel 3	PPE Thread 0
1. Write location A (IOPT[M] = ‘0’)	1. Load location B (PTE[M] = ‘1’)
	2. Any type of serialization facility that orders these loads.
2. Write location B(IOPT[M] = ‘1’, C = ‘1’, SO = ‘10’, S = ‘1’)	3. Load location A (PTE[M] = ‘1’)

If PPE thread 0 loads the new value of location B, it must also load the new value of A. On behalf of IOIF1, the IOC performs the write to memory location A before performing the write to location B.

7.7.4.3 *Example 3*

Location A is a memory location that has the caching-inhibited attribute and is therefore not in any cache. For PPE thread 0, the load uses a translated address and uses an I bit from the *PowerPC Architecture* page table entry corresponding to location A. Location A is in the noncoherent range defined by `EIB_LBAR1` and `EIB_LBAMR1` so the M bit used for both the IOIF1 and PPE thread 0 is forced to indicate the location is not coherent.

IOIF1 Virtual Channel 3	PPE Thread 0
1. Write location A (IOPT[M] = ‘0’)	1. External interrupt occurs
2. Send interrupt	2. Load location A (IOPT[M] = ‘0’, IOPT[I] = ‘1’)

After the interrupt, PPE thread 0 loads the new value of A from memory.

7.7.4.4 *Example 4*

For IOIF1 virtual channel 3, C is an IOIF address-modifier bit, M is the bit from the IOPT entry corresponding to locations A. For PPE thread 0, the load uses a translated address and uses an M bit from the *PowerPC Architecture* page table entry corresponding to location A.

IOIF1 Virtual Channel 3	PPE Thread 0
1. Write location A (IOPT[M] = ‘1’, IOPT[C] = ‘1’)	1. External interrupt occurs
2. Send interrupt	2. **lwsync** or **sync**
	3. Load location A (IOPT[M] = ‘1’)

After the interrupt, the **lwsync** or **sync** ensures that if location A was initially in the cache, the cache line invalidation due to the I/O write occurred before the load by PPE thread 0. Thus, PPE thread 0 loads the new value of A from memory.

7.7.4.5 *Example 5*

When IOPT SO equals ‘10’ or '11' and the IOIF S bit is '1' for an I/O write to memory in the CBE processor, a previous I/O write to memory in the CBE processor must be performed in order, with respect to loads performed by a PPE.

Typically, I/O writes will not be separated by an **eieio** or **sync** instruction. This requires special consideration in the system-hardware design if CBE cache-invalidate operations due to snoops are delayed. For example, if an I/O write hits a valid cache line in the PPE—which would require the cache entry to be invalidated—and the PPE delays invalidating the line, the PPE must be able to handle any future operations that hit the same line while the invalidation is still pending.

IOIF1 Virtual Channel 3 IOID 9	PPE Thread 0
1. Write location A (M = ‘1’, C = ‘1’, SO = ‘00’, S = ‘0’).	1. Load location B (M = ‘1’).
	2. Use any synchronization mechanism that causes these loads to be performed in order (see *Section 20 Shared-Storage Synchronization* on page 551).
2. Write location B (M = ‘1’, C = ‘1’, SO = ‘10’, S = ‘1’).	3. Load location A (M = ‘1’).

If the PPE loads the new value of location B, it must also load the new value of location A. Suppose location A is in the PPE’s L1 or L2 cache, but location B is not. The PPE loads the new value of B from memory. The PPE must not load the old value of A from its cache. This implies a guarantee that the cache line containing B must be invalidated before the load of A.

Below are several methods of using PowerPC assembler code for serializing loads to cacheable, coherent memory. In these methods, r1, r2, r3, and r4 are GPR locations, r2 contains the address of location B, and r1 contains the address of location A.

Method 1 (**lwsync**):

```
lwz r3,0(r2)
lwsync
lwz r4,0(r1)
```

Method 2 (**sync**):

```
lwz r3,0(r2)
sync
lwz r4,0(r1)
```

Method 3 (dependent branch and **isync**):

```
lwz r3,0(r2)
xor. r4,r3,r3
bne $-4       #branch never taken
isync
lwz r4,0(r1)
```

Method 4 (dependent operand address)

```
lwz r3,0(r2)
xor r4,r3,r3
lwz r4,r4,r1
```

7.7.4.6 *Example 6*

IOIF1 Virtual Channel 3 IOID 9	PPE Thread 0
1. Write location A (M = ‘1’, C = ‘1’, SO = ‘11’, S = ‘1’).	1. Load location B (M = ‘1’).
	2. Use any serialization mechanism that orders these loads.
2. Write location B (M = ‘1’, C = ‘1’, SO = ‘00’). The value of the S bit is irrelevant in this instance.	3. Load location A (M = ‘1’).

Even if the PPE loads the new value of location B, it may load either the new or old value of A. The IOC may perform the write to memory location A after performing the write to location B.

7.7.4.7 *Example 7*

IOIF1 Virtual Channel 3 IOID 9	PPE Thread 0
1. Read location A (M = ‘1’, C = ‘1’, SO = ‘00’, S = ‘0’).	1. Store to location B (M = ‘1’).
	2. Use any serialization mechanism that causes these stores to be performed in order.
2. Read location B (M = ‘1’, C = ‘1’, SO = ‘11’, S = ‘1’).	3. Store to location A (M = ‘1’).

If the read of A returns the new value of location A, the read of B must return the new location of B.

Any of the following PowerPC instructions can be used to serialize stores to cacheable, coherent memory: **lwsync**, **eieio**, **sync**.

7.7.4.8 *Example 8*

IOIF1 Virtual Channel 3 IOID 9	SPE 0
1. Write location A (M = ‘1’, C = ‘1’).	1. DMA from location B (M = ‘1’).
	2. Use any serialization mechanism that causes these DMA transfers to be performed in order (see the *Cell Broadband Engine Architecture*).
2. Write location B (M = ‘1’, C = ‘1’, SO = ‘10’, S = ‘1’).	3. DMA from location A (M = ‘1’).

If the SPE 0 DMA gets the new value of location B, its other DMA must get the new value of A.

7.7.4.9 *Example 9*

IOIF1 Virtual Channel 3 IOID 9	SPE 0
1. Write location A in SPE 0 LS (SO = ‘00’, S = ‘0’).	1. Read channel for SPE 0’s SPU Signal-Notification Register 1.
2. Write SPE 0’s SPU Signal-Notification Register 1 (SO = ‘10’, S = ‘1’).	2. Load from location A in SPE 0 LS.

If SPE 0 loads the new value of Signal-Notification Register 1, it must also load the new value of A.

7.7.4.10 *Example 10*

IOIF1 Virtual Channel 3 IOID 9	SPE 0
1. Write location A (M = ‘0’, C = ‘0’, SO = ‘00’, S = ‘0’).	1. Read channel for SPE 0’s SPU Signal-Notification Register 1.
2. Write SPE 0’s SPU Signal-Notification Register 1 (M = ‘0’, C = ‘0’, SO = ‘10’, S = ‘1’).	2. DMA from location A.

Location A is assumed to be in a caching-inhibited page in memory. Thus, there is no copy of memory location A in any cache when the write to location A occurs. If SPE 0 loads the new value of Signal-Notification Register 1, SPE 0 must also load the new value of A.

7.7.4.11 *Example 11*

IOIF1 Virtual Channel 3 IOID 9	SPE 0
1. Write location A (M = ‘1’, C = ‘1’, SO = ‘11’, S = ‘1’).	1. DMA from location B (M = ‘1’).
	2. Use any serialization mechanism that causes these DMA transfers to be performed in order.
2. Write location B (M = ‘1’, C = ‘1’, SO = ‘00’).	3. DMA from location A (M = ‘1’).

Even if SPE 0 DMA-transfers the new value of location B, it may transfer either the new or old value of A. The IOC may perform the write to memory location A after performing the write to location B.

7.7.4.12 *Example 12*

IOIF1 Virtual Channel 3 IOID 9	SPE 0
1. Read location A (M = ‘1’, C = ‘1’, SO = ‘00’, S = ‘0’).	1. DMA write to location B (M = ‘1’).
	2. Use any serialization mechanism that causes these DMA transfers to be performed in order.
2. Read location B (M = ‘1’, C = ‘1’, SO = ‘11’, S = ‘1’).	3. DMA write to location A (M = ‘1’).

If the read of A returns the new value of location A, the read of B must return the new value of B.

7.7.4.13 *Example 13*

IOIF1 Virtual Channel 3 IOID 9	SPE 0
1. Write location A in SPE 0 LS (M = ‘0’, C = ‘0’, SO = ‘00’, S = ‘0’).	1. Load from location B in SPE 0 LS.
2. Write location B in SPE 0 LS (M = ‘0’, C = ‘0’, SO = ‘10’, S = ‘1’).	2. Load from location A in SPE 0 LS.

If SPE 0 loads the new value of location B, it must also load the new value of A.

7.7.4.14 *Example 14*

IOIF1 Virtual Channel 3 IOID 9	SPE 0
1. Read location A in SPE 0 LS (M = ‘0’, C = ‘0’, SO = ‘00’, S = ‘0’).	1. Store to location B in SPE 0 LS.
2. Read location B in SPE 0 LS (M = ‘0’, C = ‘0’, SO = ‘11’, S = ‘1’).	2. **sync**
	3. Store to location A in SPE 0 LS.

If the read of A returns the new value of location A, the read of B must return the new value of B.

7.7.4.15 *Example 15*

A value of x‘1’ in location B indicates to an I/O device when to perform an I/O operation. The I/O device polls this location. Initially, location B has a ‘0’ value.

PPE	I/O Device
1. MMIO store to location A in I/O device.	1. Read from location B. If the I/O device received a new data value from location B, it performs step 2.
2. **sync**	
3. Store x‘1’ to location B.	2. Perform I/O operation, assuming it has new data for A.

There is no guarantee that the I/O device will receive the most recent data for location A. There is no guaranteed ordering of reads from one end of the IOIF versus writes from the other end of the IOIF.

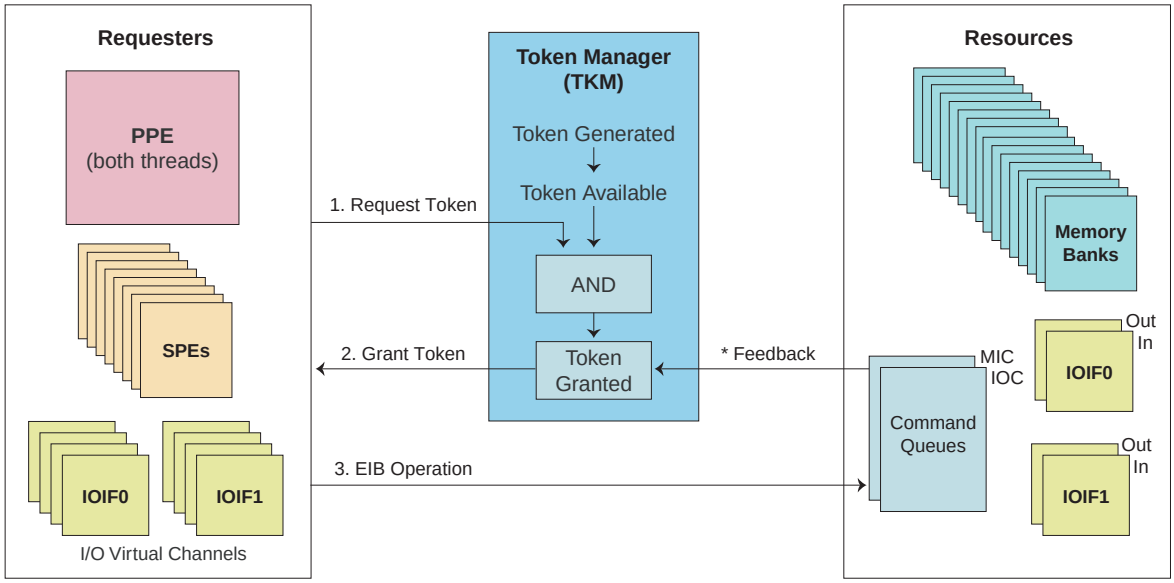
8. Resource Allocation Management

8.1 Introduction

The CBE processor's Resource Allocation Management (RAM) facility can allocate time-portions of managed *resources* (memory banks and I/O interfaces) to *requesters* (PPE, SPEs, or I/O devices). The requestors may be assigned to one of four Resource Allocation Groups (RAGs). A portion of a resource's time is represented by a *token*. The RAM facility's Token Manager (TKM) manages the amount of each resource's time that is allocated to each RAG by controlling the generation of tokens for a RAG at a programmable rate and the granting of generated tokens to a requester. To access a resource, a requester must first acquire a corresponding token from the TKM.

Figure 8-1 shows an overview of the RAM facility. Here, I/O resources are represented by the I/O Interface Controller (IOC) command queue that supports the two I/O interfaces (IOIFs), and memory resources are represented by the Memory Interface Controller (MIC) command queue. Before performing a read or write operation on the EIB that accesses a managed resource, requesters request a token for that particular resource from the TKM. The TKM generates tokens and may retain a limited number of available tokens for brief amount of time. If a token request arrives when the corresponding token is available, the token is granted, unless (for SPE and I/O requestors only) there is feedback from the MIC (for a memory bank token) or IOC (for an IOIF token) command queue to block the grant. After a token is granted, the requester can access the resource on the EIB.

Figure 8-1. Resource Allocation Overview

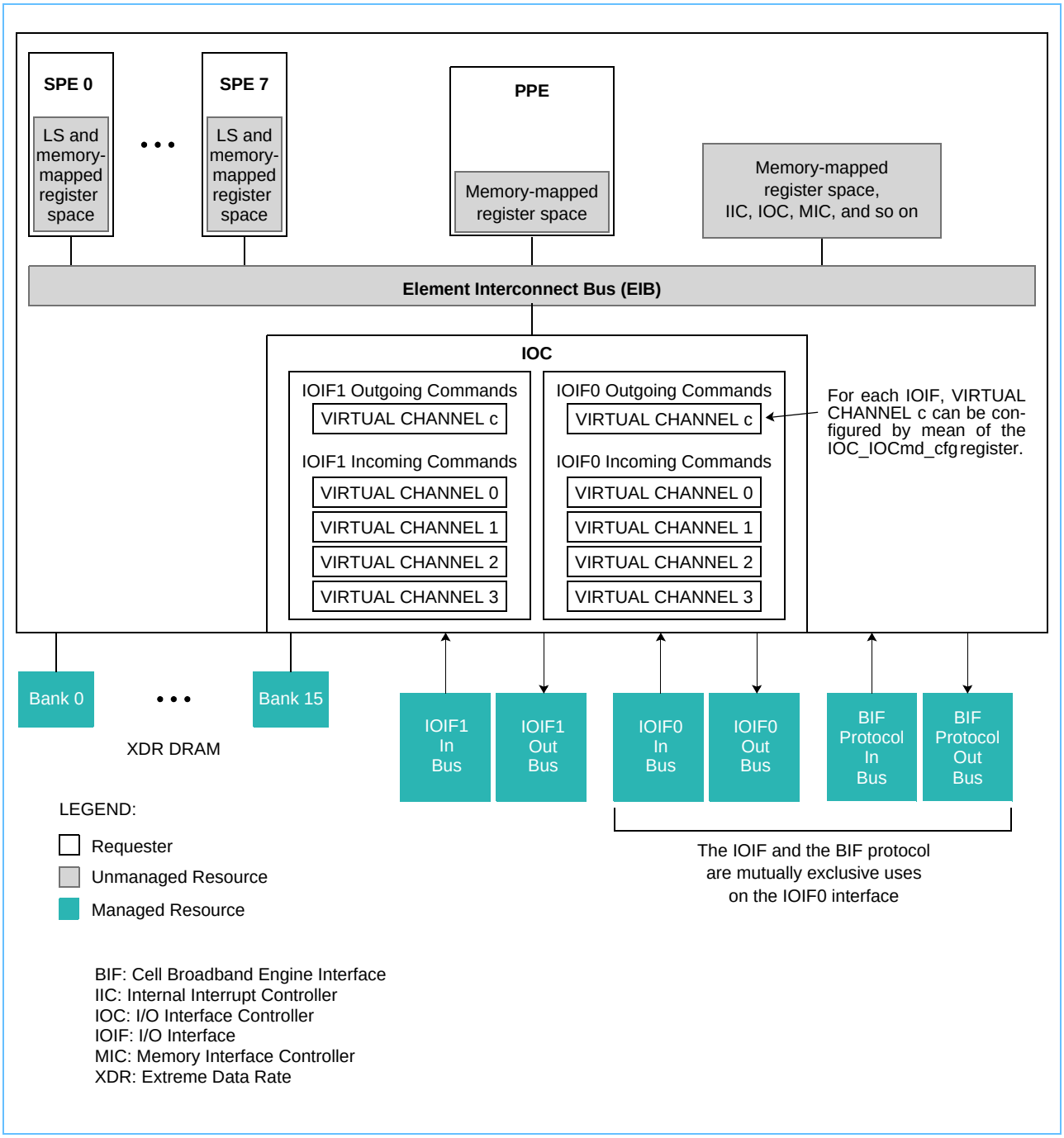


* Feedback from the IOC command queue only occurs when the IOIF (not the BIF) protocol is used.

IOC	I/O Interface Controller	PPE	Power Processor Element
IOIF	I/O Interface	SPE	Synergistic Processor Element
MIC	Memory Interface Controller	TKM	Token Manager

PPE token requests are somewhat different than shown in *Figure 8-1*, as described in *Section 8.4* on page 207. *Figure 8-2* shows a more detailed view of the requesters and resources. The SPE LSs, EIB, and CBE internal arrays and registers in memory-mapped address spaces are unmanaged resources.

Figure 8-2. Managed Resources and Requesters



8.2 Requesters

The Resource Allocation Groups (RAGs) are groups of zero or more *requesters*. These requesters are physical or virtual units that can initiate load or store requests or DMA read or write accesses. There are 17 requesters, as shown in *Figure 8-2* on page 204:

- One PPE (both threads).
- Eight SPEs.
- Four virtual I/O channels for physical interface IOIF0.
- Four virtual I/O channels for physical interface IOIF1.

Table 8-5 on page 229 lists the configuration registers used to assign requesters to RAGs.

8.2.1 PPE and SPEs

When resource allocation is enabled, each requestor belongs to one and only one RAG at any given time. However, the RAG assigned to a requestor can be changed by software over the course of time. Each RAG is identified by its Resource Allocation Group ID (RAID). The PPE and each SPE is assigned a RAID by software, as described in *Section 8.6* on page 227 and *Section 8.7* on page 231.

8.2.2 I/O

Each IOIF interface has four virtual channels, one for each of the four RAGs. For each IOIF, RAM virtual channel `r` is associated with RAID `r`, for `r` = 0 to 3. An external bridge or I/O device must be able to identify the virtual channel associated with each physical transfer on the IOIF. A virtual channel can be used for both reads and writes. This information is passed to the CBE processor with the address and command. The IOC acts as the agent on behalf of the virtual channels associated with incoming commands. The static or dynamic assignment of an I/O device to a RAG is the responsibility of the external bridge or I/O device.

8.2.2.1 *Virtual Channel Configuration*

As described in *Cell Broadband Engine Registers* specification, the MMIO IOCmd Configuration (`IOC_IOCmd_Cfg`) register defines which outgoing virtual channel is used for outgoing accesses—that is, read or write commands to the IOIF from the PPE, an SPE, or the other IOIF interface. The `IOC_IOCmd_Cfg[CVCID0]` field specifies the virtual channel for outgoing commands on IOIF0 and the `IOC_IOCmd_Cfg[CVCID1]` field specifies the virtual channel for outgoing commands on IOIF1. The outgoing virtual channel has no significance with respect to the RAM unit in the CBE processor where the command originated. However, if two CBE processors are connected by means of an IOIF, the outgoing virtual channel on one CBE processor becomes an incoming virtual channel on the other CBE processor. The incoming virtual channel is associated with a RAG in the CBE processor that is the target of the incoming command.

For an access from one IOIF to another IOIF, the virtual channel used for the incoming access on one IOIF is independent of the virtual channel used for the outgoing command on the other IOIF.

8.2.2.2 *Interchannel Delay*

Delays occur when outbound and inbound requests are competing for the same resource. For example, accesses from two virtual channels may contend for the same memory bank token; Since there is only one physical link, accesses on the link are serialized even if they are to different virtual channels.

Accesses on different virtual channels may delay each other due to:

- EIB-address or data contention.
- SPE LS contention.
- A miss in the I/O-address translation cache for one virtual channel.
- IOIF contention.

The last type of contention (IOIF) should be minimal because the IOIF resource is managed by the RAM facility.

8.3 Managed Resources

As shown in *Figure 8-2* on page 204, the RAM facility can manage up to the following 20 resources:

- *Memory banks 0 through 15*—For the PPE, both L2-cache and Non-Cacheable Unit (NCU) loads and stores.
- *IOIF0 In Bus*—IOIF or BIF protocol. For the IOIF protocol on the PPE, only non-cacheable loads; for BIF protocol on the PPE, both cacheable and non-cacheable loads.
- *IOIF0 Out Bus*—IOIF or BIF protocol. For the IOIF protocol on the PPE, only non-cacheable stores; for BIF protocol on the PPE, both cacheable and non-cacheable stores.
- *IOIF1 In Bus*—IOIF protocol. For the PPE, only non-cacheable loads.
- *IOIF1 Out Bus*—IOIF protocol. For the PPE, only non-cacheable stores.

The RAM facility allocates resources' time over long time periods. As the time period decreases, there is less certainty about allocation effects. RAM is not a solution for queueing effects when resources are highly used—in which case, latency increases. Although RAM is expected to be used for bandwidth management, RAM does not guarantee that the RAGs will get the allocated bandwidths.

RAM allocates the time that a resource is available for a RAG, without regard to whether the allocated time is actually used for requested data transfers. RAM allocates the time of resources, but does not guarantee the use by a RAG of the allocated time if this allocation exceeds the capabilities of units in the RAG or exceeds the use of the resource by the software. It is up to software to use the time of a resource in an optimal manner to achieve a desired bandwidth.

The following types of actions may help achieve optimal bandwidth:

- Writing 128 bytes in a naturally aligned 128-byte block.
- Sequential accesses that read or write equal amounts of data to all memory banks.
- Limited use of the caching-inhibited storage attribute for memory.

- Infrequent TLB misses; that is, maintain locality in loads, stores, and DMA transfers, or pin the TLB entries using RMT.
- Updating a cache line before it is displaced from the PPE L2 cache.
- Limiting the sharing of data among the PPE and SPEs or I/O devices.

8.4 Tokens

Managed resources are allocated on the basis of portions of time, represented by tokens. Before using a managed resource, a requester obtains a token for the resource, as shown in *Figure 8-1* on page 203. One token is obtained for every read or write access, of up to 128 bytes, that is wholly contained within a naturally aligned 128 bytes. Tokens are consumed when the command that required the token has been successfully transferred on the EIB.

A requester may have only one outstanding token requests per resource, with the exception that an SPE and IOC may have two outstanding token requests for the IOIF0 In and Out buses. The Token Manager (TKM) grants tokens at a programmable rate for each RAG.

8.4.1 Tokens Required for Single-CBE-Processor Systems

A requester may need multiple tokens for a single data transfer, one for each resource that its access uses. For example an I/O device on the IOIF0 interface needs two tokens to transfer data to memory, one for the specific memory bank stored to and one for the IOIF0 In data bus. *Table 8-1* shows combinations of tokens that must be obtained for operations on the EIB and IOIF interfaces in a single-CBE-processor system.

Table 8-1. Tokens Needed for EIB or IOIF Operations in Single-CBE Systems (Page 1 of 2)

Requester and Operation	Tokens Required				
	Target Memory Bank (1 of 16)	IOIF0 In	IOIF0 Out	IOIF1 In	IOIF1 Out
PPE					
Memory write of 128 bytes	1				
Memory write of 1 to 16 bytes	1 or 2[2]				
PPE or SPE					
Read from memory of 1 to 128 bytes	1				
1 to 128 byte read from I/O bridge or I/O device attached to IOIF0		1			

1. The IOC requests memory-write tokens according to the following modes, configured by means of `IOC_IOCmd_Cfg`:
   – One token regardless of the write length (1 to 128 bytes).
   – Two tokens for lengths less than 16 bytes and one token for 16 to 128 bytes (mode for memory ECC and XDR DRAMs with write-masked).
   – Two tokens for lengths less than 128 bytes and one token for 128 bytes (mode for XDR DRAMs that doesn't support write-masked).
2. For caching-inhibited writes to memory, the PPE requests 1 or 2 tokens, as specified by the BIU Configuration Ring bit "2 Token Decode for NCU Store" at Power-On-Reset (POR). If this bit is '0', one token is obtained; if this bit is '1', two tokens are obtained. Also see the `IOC_IOCmd_Cfg[SXT]` bit.

Table 8-1. Tokens Needed for EIB or IOIF Operations in Single-CBE Systems (Page 2 of 2)

Requester and Operation	Tokens Required				
	Target Memory Bank (1 of 16)	IOIF0 In	IOIF0 Out	IOIF1 In	IOIF1 Out
1 to 128 byte write to I/O bridge or I/O device attached to IOIF0			1		
1 to 128 byte read from I/O bridge or I/O device attached to IOIF1				1	
1 to 128 byte write to I/O bridge or I/O device attached to IOIF1					1
SPE					
Memory write of 16 to 128 bytes	1				
Memory write of 1 to 8 bytes	2				
IOC as Agent for I/O Device on IOIF0					
Write to LS or CBE memory-mapped registers		1			
Read from LS or CBE memory-mapped registers			1		
Write to memory of 1 to 16, 32, 48, 64, 80, 96, or 112 bytes	1 or 2[1]	1			
Write to memory of 128 bytes	1	1			
Read of 1 to 128 bytes that does not cross a 128-byte boundary	1		1		
Write to I/O bridge or I/O device attached to IOIF1		1			1
Read from I/O bridge or I/O device attached to IOIF1			1	1	
Read or write to I/O bridge or I/O device attached to IOIF0		1	1		
IOC as Agent for I/O Device on IOIF1					
Write to LS or CBE memory-mapped registers				1	
Read from LS or CBE memory-mapped registers					1
Write to memory of 1 to 16, 32, 48, 64, 80, 96, or 112 bytes	1 or 2[1]			1	
Write to memory of 128 bytes	1			1	
Read form memory of 1 to 128 bytes that does not cross a 128-byte boundary	1				1
Write to I/O bridge or I/O device attached to IOIF0			1	1	
Read from I/O bridge or I/O device attached to IOIF0		1			1
Read or write to I/O bridge or I/O device attached to IOIF1				1	1

1. The IOC requests memory-write tokens according to the following modes, configured by means of `IOC_IOCmd_Cfg`:
   – One token regardless of the write length (1 to 128 bytes).
   – Two tokens for lengths less than 16 bytes and one token for 16 to 128 bytes (mode for memory ECC and XDR DRAMs with write-masked).
   – Two tokens for lengths less than 128 bytes and one token for 128 bytes (mode for XDR DRAMs that doesn't support write-masked).
2. For caching-inhibited writes to memory, the PPE requests 1 or 2 tokens, as specified by the BIU Configuration Ring bit "2 Token Decode for NCU Store" at Power-On-Reset (POR). If this bit is '0', one token is obtained; if this bit is '1', two tokens are obtained. Also see the `IOC_IOCmd_Cfg[SXT]` bit.

8.4.1.1 *PPE Access Considerations*

The PPE has a cache with 128-byte cache lines. When software executes a load, store, **dcbt**, or **dcbtst** that is not caching-inhibited, the PPE only performs a memory read operation on the EIB if an L2 miss occurs. Any L2 miss may also cause a castout of a modified cache line, which results in a 128-byte memory write. These L2-miss memory read and write operations require a 128-byte EIB operation.

An instruction fetch may also cause an L2 miss. A load, store, or instruction fetch for which the address is translated may have a TLB miss. An L2 miss may also occur if hardware sets the page-table Change bit for a store or searches the page table for the TLB miss and the page-table entry is not in the L2 cache. A castout may also occur due to a **dcbf** instruction.

When software executes a load or a store that is caching-inhibited, the PPE does an EIB operation to memory, to an IOIF, or to the MMIO register space, depending on the destination address. The length of the load or store operation is defined by the instruction.

When software executes a store that is caching-inhibited and not guarded, the PPE may combine multiple stores to the same quadword into one store operation on the EIB. The length of this store operation can be 1, 2, 4, 8, or 16 bytes, aligned to the operand size, depending on the number of stores combined.

8.4.1.2 *SPE Access Considerations*

An SPE maintains a six-cache-line buffer (the MFC's Atomic cache). Some buffer entries are used for atomic commands and some for page-table accesses. Page-table accesses are needed when a page-table Change bit must be set or a hardware page-table search is needed for a TLB miss. When the target of an atomic command or page-table access is in this buffer, there is no read operation on the EIB. But when the target is not in this buffer, a 128-byte memory read operation occurs on the EIB. When this buffer-miss occurs, a castout of another buffer entry may occur, resulting in a 128-byte write operation on the EIB.

MFC DMA operations may need to be split into multiple EIB operations because the EIB only supports operations of 1 to 15 bytes without crossing a quadword boundary, and it only supports quadword integer multiples without crossing a 128-byte boundary.

8.4.1.3 *I/O Access Considerations*

An IOIF device can initiate read and write operations on the IOIF interface. All real-mode IOIF reads and writes and successfully translated IOIF reads and writes result in EIB operations. The IOIF device operations may need to be split into multiple IOIF operations because the IOIF interface and the EIB only support operations of 1 to 15 bytes without crossing a quadword boundary and integer multiples of quadwords without crossing a 128-byte boundary.

8.4.1.4 *Multiple Token Requests*

Some operations at the software level or I/O-device level may result in multiple token requests. For example:

- Writes to memory that are less than 16 bytes in size require a read-modify-write operation in the MIC, and this results in two token requests from the requestor—one for read and one for write.

- SPE 128-byte DMA writes to aligned caching-inhibited storage memory locations using a translated address that gets a TLB miss, which is handled by a search of the hardware page table. In this case, one or two tokens are needed for the page-table access (one for the primary hash and, potentially, one for the secondary hash) and one token is needed for the memory write.
- PPE writes to memory that experience an instruction TLB miss (up to 2 tokens), an ICache or L2 miss of the instruction (1 token), a write-address data TLB miss (up to 2 tokens), and a read (1 token). In this case, up to 6 tokens would be needed. Read latency becomes long when all memory accesses are to the same memory bank.

Because the TLB miss is not known when the access is initially attempted, and the final target resource of a DMA transfer is not known until the translation is successful, the tokens are obtained one at a time, as needed, until the real address is known. When the real address is known, at least one token is obtained, per resource used by an EIB operation, before initiating the access on the EIB.

8.4.1.5 *PPE Replacement Tokens*

PPE token requests are treated as high-priority requests by the TKM. With the exception of the PPE, tokens are not requested before the requester has a specific access to make. When the CBE processor is powered up, the PPE resets to a state where it has one token for each of the 20 possible resources at boot time. The PPE requests a replacement token in parallel with the EIB access that uses the same resource. This allows the PPE to bank one token for each resource before there is a specific access that needs the token. This reduces latency for TLB misses and L2 cache misses. If the access requires two memory tokens, the PPE may need to obtain the second token before performing the operation.

In the case of a TLB miss that requires a search of the hardware page table using the secondary hash, the accesses to the primary and secondary page-table entry group (PTEG) may occur immediately if the PPE already has tokens for these or if one of the PTEGs is in the L2 cache. However, if the page-table accesses for both hashes are to the same memory bank, the second access can only be made after the PPE has replaced the token used on the first access.

If the target is a managed resource, tokens are required for the following EIB operations: read, read with intent to modify, read with no intent to cache, write, write with kill, write with clean, and write with flush (see the *Glossary* on page 823). A memory token is consumed even though the EIB operation was satisfied by intervention and the memory bank may not actually be accessed. If speculative-read mode is enabled (`MIC_Ctl_Cnfg_0[1]` = ‘0’), a memory read may occur even though intervention occurs or the EIB command receives a retry response.

8.4.2 Operations Requiring No Token

In general, a token must be obtained by a requester for every use of memory. However, there are some exceptions to this rule. The amount of total resource time allocated should be less than the theoretical maximum resource time in order to allow for accesses that do not request tokens.

Operations that do not require tokens include:

- L2 cache writes back a modified cache line to memory in response to another requester doing one of the following:

  - EIB Write with Flush operation that hits a modified cache line (partial cache-line write due to small SPE DMA or a small or unaligned I/O device access to memory).
  - EIB reads of fewer than 128 bytes that hit a modified cache line (partial cache-line reads due to SPE DMA or I/O device access to memory; intervention on the EIB is not supported for these).
  - EIB Flush (**dcbf**) operation that hits a modified cache line.
  - EIB Clean (**dcbst**) operation that hits a modified cache line.

  For the above cases, the L2 cache writes back the modified cache line to memory without obtaining a token because there is no facility in the EIB for the PPE to obtain the RAID of the requester causing the cache line to be written back to memory. If software and I/O devices infrequently cause the above situations, the effect on the accuracy of the RAM facility is small.

- The IOC performs memory accesses for I/O-address translation. If software carefully manages the IOST cache and IOPT cache either by using the hint bits or by preloading these caches, the number of misses could be small and the effect introduced by this exception could be minimized.
- DMA transfer between the LSs of two SPEs (the LS is an unmanaged resource).
- Accesses to memory-mapped registers within the CBE processor.
- All EIB requests that do not require a data transfer, including **sync**, **tlbie**, **tlbsync**, **eieio**, and **larx**.

8.4.3 Tokens Required for Multi-CBE-Processor Systems

The RAM facility is supported in a limited manner for multiple CBE processors connected through the IOIF0 interface using the BIF protocol. In a multi-CBE system, the use of resources in or attached to a CBE processor (also referred to as the *local CBE processor*) are managed by the TKM within that CBE processor for accesses by requesters within that CBE processor. The use of memory and IOIF1 resources in or attached to a CBE processor by requesters in the other CBE processor are not directly managed. Such resources use can only be indirectly managed by managing the use of the BIF protocol. Resource allocations for IOIF0 effectively become allocations for the BIF protocol, although configuration bits in the PPE, SPEs, and IOC can be set so these units do not depend on obtaining tokens.

The PPE, SPEs, and IOC have configuration registers that recognize the real addresses corresponding to:

- Memory in the local CBE processor.
- IOIF1 attached to the local CBE processor.
- 8-MB MMIO register space within the local CBE processor.

Addresses associated with IOIF0 or the BIF protocol are addresses that do not correspond to one of these three local spaces. In a multi-CBE-processor environment, this allows all addresses associated with resources in or attached to the other CBE processor to be identified as using the BIF protocol. Because the BIF-protocol IOIF resource is used by these accesses, by the command and response phase of any coherent read or write, and by command-only transac-

tions, the availability of the IOIF resource for read and write data cannot be definitively stated. However, an IOIF0 token is required for BIF-protocol reads and writes that might cause data to be transferred.

8.5 Token Manager

The Token Manager (TKM) is shown in *Figure 8-1* on page 203 It receives token requests, generates tokens at a software-programmable rate, and grants tokens to requestors when the tokens are available. In each clock cycle, each physical requester might request a token. Token requests are latched by the TKM and potentially granted at a later time.

8.5.1 Request Tracking

The TKM tracks 17 virtual requesters (*Section 8.2* on page 205), each with up to 20 resource targets. The TKM also tracks the priority of requests for the SPEs. For PPE and SPE requests, the TKM records the RAID for the RAG. Internally, the TKM keeps track of requests by RAG. For each RAG, the TKM records nine high-priority requests (PPE and SPE 0:7) for each resource and 10 low-priority requests (SPE 0:7 and IOC 0:1) for each resource. The TKM allows both IOIF and memory token requests on both IOC0 and IOC1.The TKM uses round-robin pointers for selecting among the high- and low-priority requests within each RAG. For more information, see *Section 8.5.5.4 Memory-Token Grant Algorithm* on page 217 and *Section 8.5.6 I/O Tokens* on page 218.

8.5.2 Token Granting

Figure 8-3 on page 213 shows the organization of the TKM with respect to token-granting. For each RAG, the TKM uses Allocation Registers and Rate Decrementers to determine how often and exactly when to grant tokens, and it uses Token-Available Latches to record when tokens are available for the RAG. It has round-robin pointers per priority, per resource, and per RAG to track which of multiple requesters in a RAG should be granted a token. For tokens that are not used by a RAG and that may be granted to other requesters in other RAGs—the Unallocated RAG (RAG U) described in *Section 8.5.3* on page 214—the TKM Priority Register (`TKM_PR`) defines the priority of other RAGs in obtaining these unused available tokens.

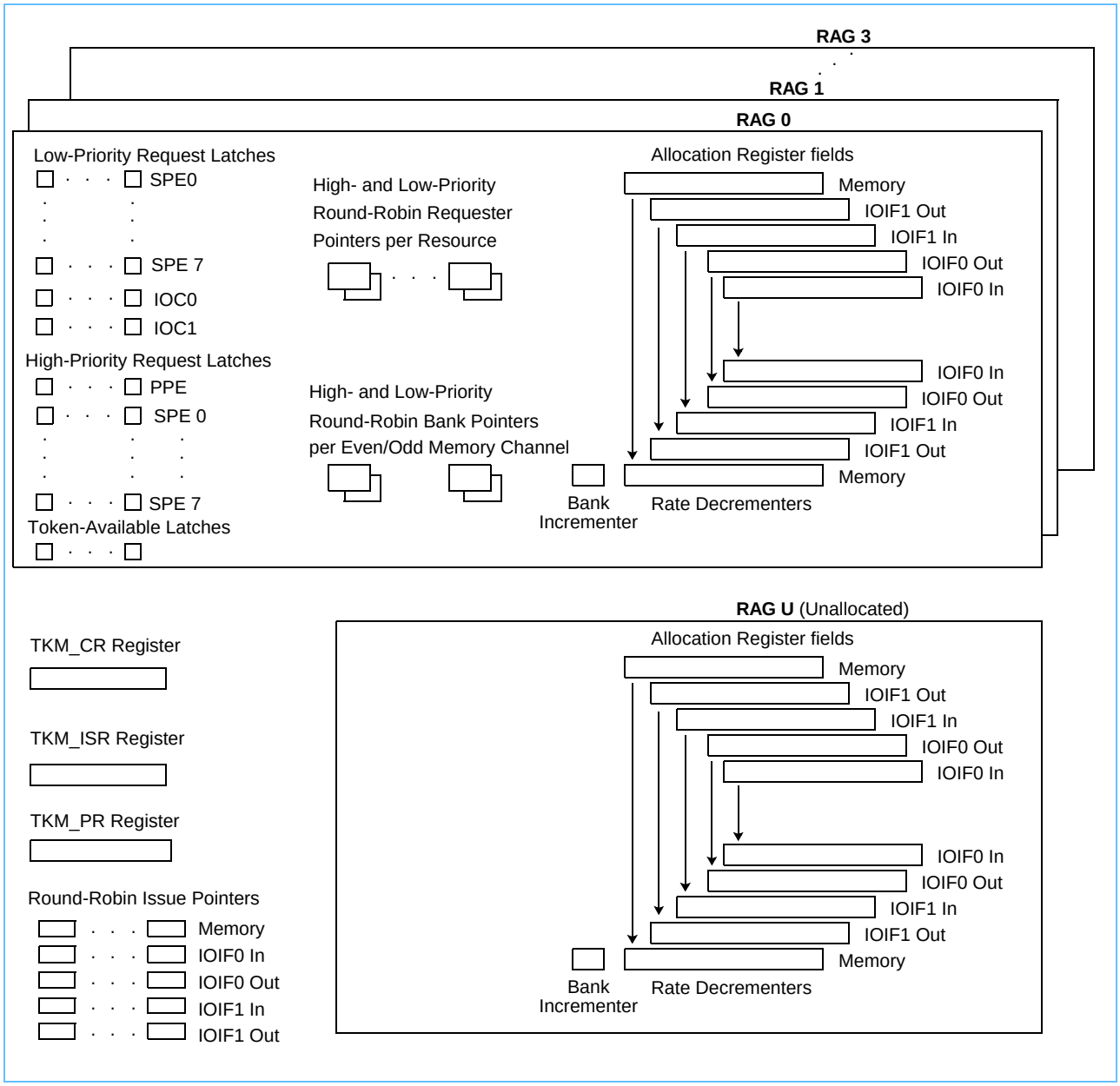
`TKM_CR[44:47]` enables the granting of tokens to RAG 0:3, respectively. When the bit for a RAG is disabled, no tokens are granted for requesters assigned to the RAG. If a particular RAG is not used, software can disable the RAG to save power by clearing the RAG-enable in `TKM_CR[44:47]`. However, disabling the RAG does not prevent token-generation within the RAG; token generation is disabled by disabling the Rate Decrementer, which is done by clearing the appropriate bit in `TKM_CR[55:58]` for the desired RAG. `TKM_CR[54]` controls token generation for RAG U (see *Section 8.5.3* on page 214). Any token generated by a RAG but not granted to a requestor can eventually be classified as unused, as described in *Section 8.5.7 Unused Tokens* on page 218. Unused tokens may be shared and granted to another RAG if enabled by `TKM_CR[51]`.

In order to grant tokens at a programmable rate, the Allocation Registers define the period of time between token-generation for a RAG. The TKM uses the TKM Memory Bank Allocation (`TKM_MBAR`), TKM IOIF0 Allocation (`TKM_IOIF0_AR`), and TKM IOIF1 Allocation (`TKM_IOIF1_AR`) MMIO registers for this purpose. The Rate Decrementers track the periods of time. Rate Decre-

menters cannot be directly loaded by software. These decrementers are decremented at a rate specific to the resource and at a rate established by a programmable prescaler specified by an Allocation Register. When a Rate Decrementer is ‘0’ and is to be decremented again, the Rate Decrementer expires. Not only must the Rate Decrementer reach ‘0’, it also must be decremented again, at which time it is loaded again, based on the allocation.

When a Rate Decrementer for memory expires, a token becomes available for the associated RAG and the 3-bit interval value from the appropriate Allocation Register field is loaded into the Rate Decrementer. The most-significant bit of the interval value loaded into a Rate Decrementer is an implied '1', with the following exception for memory: when `TKM_MBAR[33]` = ‘0’ and a RAG's Prescaler value for memory is '0', a '0' is loaded into the RAG's Rate Decrementer for memory.

Figure 8-3. Token Manager



8.5.3 Unallocated RAG

In addition to RAGs 0:3, there is a RAG called Unallocated, or RAG U. There are no requesters in this RAG. Tokens are generated for this RAG using a Rate Decrementer and Allocation Register similar to other RAGs. Because there are no RAG U requesters, all tokens generated for RAG U become unused tokens that may be granted to other requesters in other RAGs. The `TKM_PR` register defines the other RAGs' priority in obtaining Unallocated tokens. See *Section 8.5.7 Unused Tokens* on page 218 for an explanation of the unused tokens.

RAG U can take care of occasional spikes in token request rates—for example, if RAG 0 usually needs only 10% of memory bandwidth but occasionally needs 20% for a brief time. In such a case, it might be best to allocate only 10% to RAG 0, but allow RAG U tokens be given to RAG 0 to cover the occasional difference. RAG U is also a buffer that is taken from circulation because the MIC queues may fill up due to random-access patterns from RAGs. Unallocated tokens are sensitive to feedback that represents the use of a resource. See *Section 8.5.12 Feedback from Resources to Token Manager* on page 226. RAG U can also be used to allocate resource time that is more likely to be used by multiple RAGs.

An Unallocated token is only given to a RAG that has an outstanding request at the instant the token is generated, whereas a token generated for RAG 0:3 is made available to the RAG for which it is generated if there is no such token already available for the same resource.

8.5.4 High-Priority Token Requests

SPEs indicate priority along with their token requests. High-priority requests by the SPE are only used for memory accesses. All PPE token requests have an implicit high priority, even though the PPE does not drive the high-priority signal. PPE token requests have high priority for IOIF0 In and Out, IOIF1 In and Out, and memory tokens. SPE high-priority token requests are only enabled if the Priority Enable bit, `TKM_CR[PE]`, is '1'. However, if `TKM_CR[PE]` is '0', all SPE token requests are treated as low-priority.

Except for `TKM_CR[PE]`, software has no control on high-priority requests. Instead, hardware controls which token requests are high-priority. If there are multiple high-priority requests for a resource from requesters in a RAG, their requests are granted tokens on a round-robin basis. Low-priority requests from requesters in a RAG are only granted tokens when there are no high-priority requests from requesters in the RAG for the same resource. If there are no high-priority requests from requesters in a RAG for a resource and there are multiple low-priority requests from requesters in the RAG for the resource, their requests are granted tokens on a round-robin basis.

An SPE drives the priority signal high when it requests a token for any of the following reasons:

- The **getllar**, **putllc**, or **putlluc** MFC atomic update commands.
- A TLB miss.
- A memory write of a modified cache line.

If a requester already has a low-priority token request for a resource outstanding, and it has another access to the same resource that warrants high priority, the requester cannot issue another token request for the same resource.

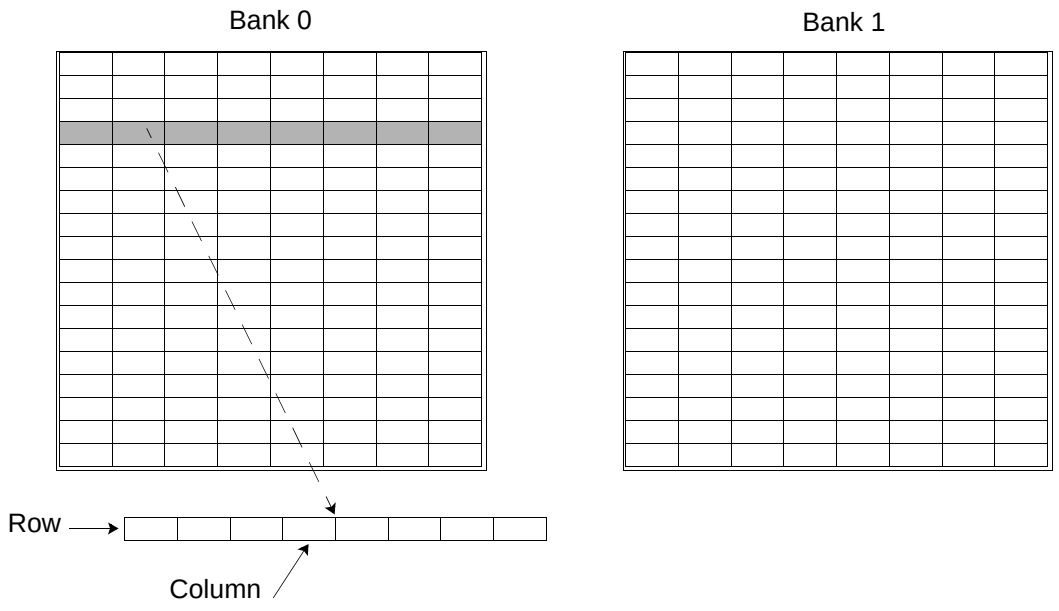
8.5.5 Memory Tokens

8.5.5.1 *Memory Banks*

Inside each Extreme Data Rate (XDR) DRAM device, there are independent memory banks. Each memory bank can handle only one operation at a time. The operations they handle are reads, writes, and refreshes.

The MIC supports 4, 8, and 16 memory banks on each of the two XDR memory interfaces (channels). Because the memory interfaces are independent, the number of effective memory banks supported is double the number each XDR DRAM device contains when both channels are used. *Figure 8-4* illustrates the differences between rows, columns, and banks. Two banks are shown in this logical representation of the internal organization of an XDR DRAM. One row of Bank 0 is highlighted.

Figure 8-4. Banks, Rows, and Columns



If there are 16 memory banks, real addresses are interleaved across the 16 memory banks on a naturally aligned 128-byte basis. Bank 0 holds the first 128-byte block and every sixteenth 128-byte block after the first. Bank 1 holds the second 128-byte block and every sixteenth 128-byte block after the second, and so forth. If there are eight memory banks, real addresses are interleaved across the eight memory banks on a naturally aligned 128-byte basis. Thus, regardless of the actual number of memory banks, memory accesses are managed as if there are 16 logical banks. If there are only eight banks, each physical bank is treated as two logical banks from a RAM-facility perspective.

When an XDR DRAM device is busy doing an access to a bank, other accesses to the same bank are delayed until that access is complete. The MIC starts accesses to other banks if there are commands for those banks queued inside the MIC and those banks are free. Therefore, to

maximize performance, addresses should be spread across all banks. The RAM facility understands memory banks and allocates tokens based on this same definition. The more banks available, the more parallel memory operations can occur at the same time.

8.5.5.2 *Memory-Bank Tokens*

A pair of memory-bank tokens, one even and one odd, are generated when the Rate Decrementer would decrement below the value of ‘0’. These generated bank tokens become available to be granted to requesters in the RAG for which they are generated. The choice of banks for which a token becomes available is based on the value of the Bank Incrementer of a RAG. In the same clock cycle in which a bank token is generated, the Bank Incrementer is incremented by one.

8.5.5.3 *Memory-Channel Tokens*

When there have been no requests for tokens for a long period of time, the TKM can accumulate up to one token for each managed resource, thus up to 16 tokens for all memory banks. To reduce the rate at which memory tokens are granted if there is a sudden burst of requests for the available bank tokens, the TKM also maintains internal *memory-channel 0 tokens* and *memory-channel 1 tokens*. These memory-channel tokens represent XDR DRAM bus time and are used to pace the rate at which bank tokens are granted.

Memory-channel tokens also allow accesses to be concentrated in a subset of the banks while preventing the XDR DRAM bus from being overused. For example, for certain XDR DRAM t_{RC} values, memory bandwidth could be fully used even though all the accesses went to half the banks. The memory-channel tokens are maintained internal to the TKM and are not tokens that are requested or granted to requesters. The TKM only grants a bank token if both a bank token and a corresponding memory-channel token are available. For even-numbered banks, a memory channel 0 token is needed. For odd-numbered banks, a memory channel 1 token is needed.

See the Memory Interface Controller (MIC) MMIO Registers section in the *Cell Broadband Engine Registers* document for configuration information about XDR DRAM t_{RC}. To allow for different XDR DRAM t_{RC} values, the Mode for Token-Generation bit, `TKM_CR[MT]`, defines the relative rate at which memory-channel tokens are generated versus memory-bank tokens. Three rates are possible:

- When `TKM_CR[MT]` = ‘00’, memory-channel tokens are made available at the same rate that bank tokens are made available. In this case maximum memory bandwidth can be achieved only if all the banks are equally accessed.
- When `TKM_CR[MT]` = ‘01’, memory-channel tokens are made available at 75% of the rate that bank tokens are made available. Thus maximum memory bandwidth can be achieved even if only three quarters of the banks are accessed.
- When `TKM_CR[MT]` = ‘10’, memory-channel tokens are made available at half the rate that bank tokens are made available. When `TKM_CR[MT]` = ‘11’, operation of the TKM is undefined.

Table 8-2 summarizes when memory bank and memory-channel tokens become available for the modes defined by `TKM_CR[MT]`.

Table 8-2. Memory Tokens Available Based on TKM_CR[MT]

Bank Incrementer	Memory Tokens That Become Available When Rate Decrementer Decrements Below 0				
	Bank Tokens	Memory Channel 0 and Memory Channel 1 Tokens			
		MT = ‘00’	MT = ‘01’	MT = ‘10’	MT = ‘11’
0	0 and 1	memory channel 0, and memory channel 1			Undefined
1	2 and 3	memory channel 0, and memory channel 1	memory channel 0, and memory channel 1	memory channel 0, and memory channel 1	Undefined
2	4 and 5	memory channel 0, and memory channel 1	memory channel 0, and memory channel 1		Undefined
3	6 and 7	memory channel 0, and memory channel 1	memory channel 0, and memory channel 1	memory channel 0, and memory channel 1	Undefined
4	8 and 9	memory channel 0, and memory channel 1			Undefined
5	10and 11	memory channel 0, and memory channel 1	memory channel 0, and memory channel 1	memory channel 0, and memory channel 1	Undefined
6	12 and 13	memory channel 0, and memory channel 1	memory channel 0, and memory channel 1		Undefined
7	14 and 15	memory channel 0, and memory channel 1	memory channel 0, and memory channel 1	memory channel 0, and memory channel 1	Undefined

Although the memory-channel tokens prevent bank tokens from being granted in big bursts, it also implicitly requires that tokens be requested at a more steady rate. For example, if the PPE was alone in one RAG with 100% allocation of memory, a pair of memory-channel tokens are made available every interval of time as defined by the `TKM_MBAR` register. An even- and odd-bank token must be requested in every such interval, or some of these tokens would become unused.

Memory-bank tokens are made available for banks in increasing order. If software fetches data structures from memory in this order, the result might be better use of available tokens.

8.5.5.4 *Memory-Token Grant Algorithm*

The following description of memory-token granting applies to one RAG, the requests in the RAG, and the round-robin pointers in that RAG:

For each memory bank and for each RAG, there is a round-robin pointer to select the next high-priority request in the RAG for the bank, if there is more than one high-priority request active. Likewise, there is a round-robin pointer to select the next low-priority request in the RAG for the bank, if there is more than one low-priority request active. A request is only eligible to be granted a token if the round-robin pointer points to the request.

Tokens for odd banks are granted in a manner similar to even banks, except that a memory-channel 1 token must be available and odd-bank pointers are used to select the bank and requester.

8.5.6 I/O Tokens

For each RAG 0:3 and RAG U, there are separate Rate Decrementers for each IOIF bus direction, in and out. This is necessary because each IOIF and each direction can have different bus widths. When an IOIF0 In Rate Decrementer, IOIF0 Out Rate Decrementer, IOIF1 In Rate Decrementer, or an IOIF1 Out Rate Decrementer of a RAG is ‘0’ and is about to be decremented again, a token becomes available for the corresponding bus.

The following description applies to one RAG, the requests in the RAG, and the round-robin pointers in that RAG:

The token-grant facilities for all IOIF buses (IOIF0 In, IOIF0 Out, IOIF1 In, and IOIF1 Out) are alike. The PPE is the only high-priority requester for IOIF buses. For each IOIF bus and for each RAG, there is a round-robin pointer to select the next low-priority request in the RAG for the IOIF bus, if there is more than one low-priority request active. A request is only eligible to be granted a token if the round-robin pointer points to the request.

If an IOIF bus token is available and there is either a high or low-priority eligible request for the IOIF bus, the IOIF bus token is granted in the following manner.

- If there is a high-priority PPE request for the token, the IOIF bus token is granted to the PPE.
- If there is no high-priority eligible request, the IOIF bus token is granted to the requester whose request is pointed to by the low-priority request round-robin pointer for the IOIF bus.
- The Token-Available Latch for the IOIF bus is reset. The request latch corresponding to the requester granted the token is reset.

8.5.7 Unused Tokens

If there is no token for a resource already available for a RAG 0:3 whose Rate Decrementer generates a token, the new token is made available only to that RAG and remains available until the token is granted to the RAG. If there is a token for the resource already available to that RAG, the new token is called an *unused token*.

This unused token is available to other RAGs that have outstanding requests for the resource, if this function is enabled by the Unused Enable bit, `TKM_CR[UE]`. Only RAGs with outstanding requests are eligible for an unused token. If there are multiple RAGs with outstanding requests, the token is given to the RAG with the highest priority, per the `TKM_PR` register, for the specific RAG with the unused token.

Tokens issued from the staging buffer are ready to be used. The tokens may be granted if an outstanding token request exists for that resource, retained in the Token-Available Latch for that resource if there is no existing token, or shared with another RAG as an unused token under the guidelines given above.

8.5.8 Memory Banks, IOIF Allocation Rates, and Unused Tokens

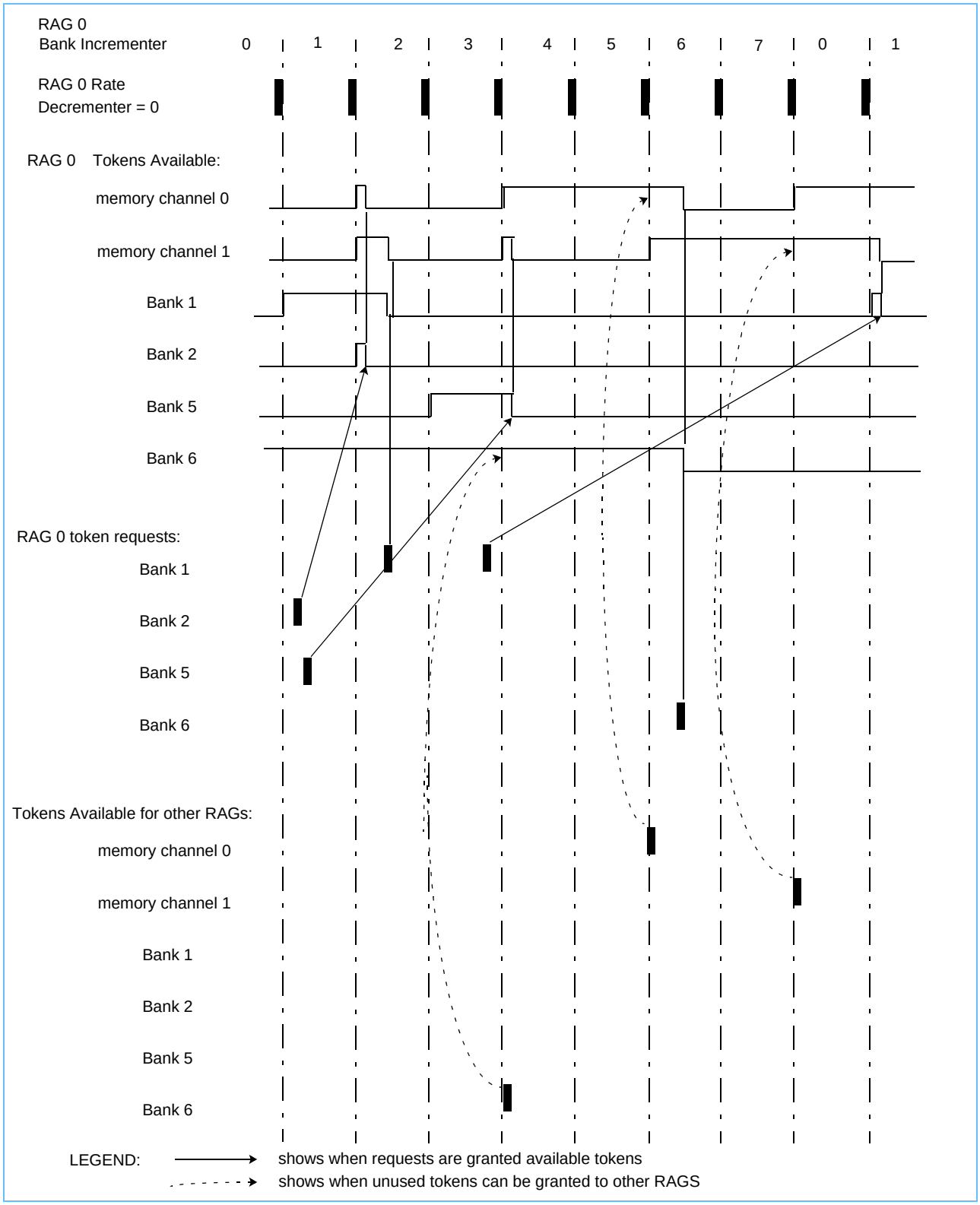
Tokens for memory banks are described in *Section 8.5.5* on page 215, tokens for I/O tokens are described in *Section 8.5.6* on page 218, and unused tokens are described in *Section 8.5.7* on page 218. *Section 8.5.10.1 Memory-Allocation* on page 223 and *Section 8.5.10.2 IOIF Allocation* on page 223 describe the calculation of allocation rates for memory and IOIF.

The sum of all RAGs' allocation rates for any given resource should not exceed 100% of the resource. Otherwise, tokens might be lost in the token generator and the resource could be overcommitted. If the resource is overcommitted, the exact amount of the resource that can be used by any given RAG is unpredictable.

8.5.9 Token Request and Grant Example

Figure 8-5 on page 220 shows an example of token requests and grants when `TKM_CR[MT]` = '10'. Bold vertical lines indicate the TKM clock cycle(s) when the RAG 0 Rate Decrementer has a value of '0'.

Figure 8-5. Memory-Token Example for TKM_CR[MT] = '10'



For the example in *Figure 8-5* on page 220, the Prescaler for RAG 0 (`P0`) field of the TKM Memory Bank Allocation Register (`TKM_MBAR`) is ‘1’, so the RAG 0 Rate Decrementer contains a ‘0’ value for two `NClk`/2 cycles. After two cycles, the RAG 0 Rate Decrementer, instead of being decremented, is reloaded from the Interval value for RAG 0 (`I0`) field of the `TKM_MBAR` register, and the RAG 0 Bank Incrementer is incremented.

With `TKM_CR[MT]` = ‘10’, tokens for memory channel 0 and 1 become available when the RAG 0 Bank Incrementer changes from an odd value to an even value. In the cycle when the RAG 0 Rate Decrementer is reloaded from the `TKM_MBAR` register, a token becomes available for two banks based on the RAG 0 Bank Incrementer value. When the Bank Incrementer value changes from i to i+1, tokens for banks `2i` and `2i` + 1 become available. For this example, `TKM_CR[UE]` = ‘1’ to allow sharing of unused tokens.

The following explanation is divided into time periods that correspond to the RAG 0 Bank Incrementer value:

- RAG 0 Bank Incrementer = 1:
  - When RAG 0 Bank Incrementer changes from 0 to 1, a token for Bank 1 is generated and the Token-Available Latch is set for this bank. Although not shown in *Figure 8-5*, a token for Bank 0 would also become available.
  - During this period, there is a RAG 0 request for Bank 2. The tokens for Bank 2 and memory channel 0 are not available. Therefore, a token for Bank 2 cannot be immediately granted.
  - During this period, there is also a RAG 0 request for Bank 5. The tokens for Bank 5 and memory channel 1 are not available. Therefore, a token for Bank 5 cannot be immediately granted.
- RAG 0 Bank Incrementer = 2:
  - When RAG 0 Bank Incrementer changes from 1 to 2, tokens for Bank 2, memory channel 0 and memory channel 1 are generated, and the Token-Available Latches are set for Bank 2, memory channel 0, and memory channel 1. Although not shown in *Figure 8-5*, a token for Bank 3 would also become available.
  - Because tokens for both Bank 2 and memory channel 0 are now available, a token for Bank 2 is granted to satisfy the outstanding RAG 0 request from the previous period, and the RAG 0 tokens for Bank 2 and memory channel 0 are no longer available for further grants until the TKM generates new tokens for these resources, based on the RAG 0 Bank Incrementer and Rate Decrementer. The token for memory channel 0 is not explicitly granted, but the available token is internally consumed by the TKM as a prerequisite for granting a token for Bank 2.
  - Tokens for both Bank 1 and memory channel 1 are available, so a token for Bank 1 is granted as soon as a RAG 0 requester requests this token in the period of time when the RAG 0 Bank Incrementer is 2. The Token-Available Latches for Bank 1 and memory channel 1 are reset.
- RAG 0 Bank Incrementer = 3:
  - When the RAG 0 Bank Incrementer changes to 3, a token for Bank 5 becomes available. Although not shown in *Figure 8-5*, a token for Bank 4 would also become available.

  - Although the token for Bank 5 is available and there is an outstanding request from the previous period, a token for Bank 5 cannot be granted because there is no available token for memory channel 1.
  - During this period, there is an RAG 0 request for Bank 1. Because the tokens for Bank 1 and memory channel 1 are not available, the request cannot be immediately granted.
- RAG 0 Bank Incrementer = 4:
  - When the RAG 0 Bank Incrementer changes to 4, tokens for Bank 6, memory channel 0 and memory channel 1 become available. Although not shown in *Figure 8-5*, a token for Bank 7 would also become available.
  - Because the tokens for both memory channel 1 and Bank 5 are now available, the outstanding RAG 0 request for Bank 5 (from period 1) is granted a token for Bank 5.
  - Because the token for Bank 6 was already available, this newer token (unused token) becomes available for another RAG with an outstanding request for Bank 6.
- RAG 0 Bank Incrementer = 5:
  - When the RAG 0 Bank Incrementer changes to 5, although not shown in *Figure 8-5*, tokens for Banks 8 and 9 would become available at this time.
- RAG 0 Bank Incrementer = 6:
  - When the RAG 0 Bank Incrementer changes to 6, tokens for memory channel 0 and 1 become available. Although not shown in *Figure 8-5*, tokens for Banks 10 and 11 would become available at this time.
  - Because the token for memory channel 0 was already available, this newer token becomes available for another RAG with an outstanding request needing a token for memory channel 0.
  - When a RAG 0 request for a token for Bank 6 occurs, the token is immediately granted because tokens for both Bank 6 and memory channel 0 are available.
- RAG 0 Bank Incrementer = 7:
  - When the RAG 0 Bank Incrementer changes to 7, although not shown in *Figure 8-5*, tokens for Banks 12 and 13 would become available at this time.
- RAG 0 Bank Incrementer = 0:
  - When the RAG 0 Bank Incrementer changes to 0, tokens for memory channel 0 and 1 become available. Although not shown in *Figure 8-5*, tokens for Banks 14 and 15 would also become available.
  - Because the token for memory channel1 was already available, this newer token becomes available for another RAG with an outstanding request needing a token for memory channel 1.
- RAG 0 Bank Incrementer = 1:
  - When the RAG 0 Bank Incrementer changes to 1, a token for Bank 1 becomes available. Although not shown in *Figure 8-5*, a token for Bank 0 would also become available.
  - Because there are available tokens for both Bank 1 and memory channel 1, the outstanding request for Bank 1 (from period 3) is granted a token for Bank 1.

8.5.10 Allocation Percentages

8.5.10.1 *Memory-Allocation*

The total memory-allocation percentage equals the sum of percentages of all RAGs. The percentage of memory-resource time allocated to a single RAG can be calculated using the following equations, in which the `I` and `P` variables are the Interval and Prescaler values from the appropriate fields of the `TKM_MBAR` register, and `MT` is the Mode-for-Token-generation field in the `TKM_CR` register.

These equations assume that `TKM_CR[MT]` is configured consistently with XDR DRAM t_{RC}. The maximum value of the t_{RC} for reads and writes must be used in this calculation. CFM and CFMN are differential input clock signals to the CBE processor. See the *Cell Broadband Engine Datasheet for DD 3.O and DD 3.1* and the *Rambus XDR Architecture (DL-0161)* document for more information about CFM. The number of memory channels in the equations is the number of memory channels in the CBE processor that are used to attach XDR DRAM.

`NClk` is the CBE Core Clock. The EIB, TKM, and portions of MIC and BEI run at half this frequency (`NClk`/2). See *Section 13 Time Base and Decrementers* on page 375 and the *Cell Broadband Engine Datasheet for DD 3.O and DD 3.1* for more information about the CBE clocks the available bandwidth for XDR DRAM.

The equations are:

```
Allocated% = 100 * allocated bandwidth capability / available bandwidth

Allocated bandwidth capability [bytes/s] =
    128 [bytes/memory-channel-token] *
    (4-MT)/2 [memory-channel-tokens/decrementer-expiration] *
    1/(<NClk/2 period> [s/EIBcycle] *
    (I+1+8*((P>0) OR TKM_MBAR[33]))*(2^P) [EIBcycles/decrementer-expiration])
```

8.5.10.2 *IOIF Allocation*

The total IOIF-allocation percentage equals the sum of percentages of all RAGs. The percentage of IOIF-bus resource time allocated to a single RAG can be calculated using the following equations, in which the `I` and `P` variables are the Interval and Prescaler values from the appropriate fields of the `TKM_IOIF0_AR` or `TKM_IOIF1_AR` registers. The IOIF0 and IOIF1 allocation percentages are different for the same values of `I` and `P`.

The equations are:

```
Allocated% = 100 * allocated bandwidth capability / available bandwidth

IOIF0 allocated bandwidth capability

    = (128 bytes / token) * (1 token per Decrementer expiration) * (1 Decrementer
    expiration / (NClk/2 period * (9 + I) * 2^P))

    = 128 bytes * NClk/2 frequency / ((9 + I) * 2^P)
```

```
IOIF1 allocated bandwidth capability = 128 bytes * NClk/2 frequency / ((9 + I) * 2^(p+2))
```

8.5.11 Efficient Determination of TKM Priority Register Values

A method is described here for calculating the TKM Priority Register (`TKM_PR`) fields using the *PowerPC Architecture* instructions.

There are enables in the `TKM_PR` register for each RAG to obtain tokens from every other RAG. For example, the `TKM_PR` register has four enables for obtaining unused tokens from RAG U—one enable for each of RAG 0:3. There are three enables for obtaining unused tokens from RAG 0—one each for RAG 1:3.

The values for the `TKM_PR` fields can be calculated without a large number of instructions and without calculating each individual bit as a separate calculation. An example is used to explain the algorithm for determining priorities for RAGs 0:3 in obtaining unused tokens from RAG U. For this example, the priority for getting RAG U tokens is given as the following order, from highest- to lowest-priority:

```
RAG1 > RAG0 > RAG3 > RAG2
```

RAG1 has highest priority and RAG2 has lowest priority.

Table 8-3 shows the universal set of binary OR and AND values for all RAG priorities. These values are used in the algorithm steps that follow. The values would be also used every time priority is determined, not only for this example but for all RAG priorities, but the values would be used by the algorithm steps in a different order, in which the order is based on the relative priority of the various RAGs.

Table 8-3. OR and AND Values for All RAG Priorities

Binary Value	Set Priority OR and AND value
111000	RAG 0 set priority OR_value
111111	RAG 0 set priority AND_value
000110	RAG 1 set priority OR_value
011111	RAG 1 set priority AND_value
000001	RAG 2 set priority OR_value
101011	RAG 2 set priority AND_value
000000	RAG 3 set priority OR_value
110100	RAG 3 set priority AND_value

This three-step algorithm is used for determining priority for RAGs 0:3 in obtaining RAG U unused tokens (with all 6 digits necessary), and for determining the priority for one RAG obtaining unused tokens from another RAG (with only the last three digits used). When the steps are used for determining priority for unused tokens from RAG 0:3, the number of RAGs used in Step 2 is one fewer, and only a 3-bit value is produced in the least-significant bits of GPR *r* by

Steps 1 and 2. Similar to Step 3, this 3-bit value should be rotated and inserted into the 3 bit positions in `TKM_PR` for the appropriate RAG's Unused tokens. Thus, for RAG 0 unused tokens, the 3-bit value should be rotated to bits 43:45 of the data value to be stored to `TKM_PR`.

The steps for determining priority are:

1. Clear a General-Purpose Register (GPR) *r* in the PPE to hold the results for the RAG U priority fields, where the least-significant 6 bits of the GPR will eventually hold the 6-bit value corresponding to `TKM_PR[34:39]`.
2. For each of three RAGs, starting with the next-to-last RAG in the priority order (that is, the second from lowest priority) and working backward to the RAG with the highest priority order:
   a. Logically OR GPR *r* with the OR_value for the RAG.
   b. Logically AND GPR *r* with the AND_value for the RAG.
3. GPR *r* now has the value of the priority fields for RAG U, so rotate and insert this value into a GPR that will hold the data value to be stored to `TKM_PR` register.

Applying the above three steps to the example:

1. Clear the GPR `r` bits. The least-significant 6 bits of this GPR, corresponding to the priority fields, have the following values:

   `0 0 0 0 0 0`

2. Apply this step for 3 RAGS:
   a. GPR `r` least-significant 6 bits after RAG 3 values are ORed and ANDed:

      `0 0 0 0 0 0`

   b. GPR `r` least-significant 6 bits after RAG 0 values are ORed and ANDed:

      `1 1 1 0 0 0`

   c. GPR `r` least-significant 6 bits after RAG 1 values are ORed and ANDed:

      `0 1 1 1 1 0`

3. GPR `r` now has the value to be rotated and inserted into a GPR that holds the data value to be stored to `TKM_PR` register.

The 6-bit value being determined for RAG U unused-token priorities encompasses three distinct fields of the `TKM_PR` register—`RU0Vr`, `RU1Vr`, and `RU2V3`. When the algorithm is used to find the 3-bit values for determining priorities from unused tokens from RAG 0:3, those three bits encompass two distinct fields of the `TKM_PR` register.

The following *PowerPC Architecture* instructions are used to determine the values to be loaded into the RU0Vr, RU1Vr, and RU2V3 fields of the `TKM_PR` register for this example. GPR 1 is used for calculating the value to be rotated/inserted into GPR 2, which can be used to collect values calculated for other fields before GPR 2 is finally written to the `TKM_PR` register.

```
xor 1,1,1                    * clear GPR 1
ori 1,1,b'000000'            * logically OR GPR 1 with RAG 3 OR_value
andi. 1,1,b'110100'          * logically AND GPR 1 with RAG 3 AND_value
ori 1,1,b'111000'            * logically OR GPR 1 with RAG 0 OR_value
```

```
andi. 1,1,b'111111'          * logically AND GPR 1 with RAG 0 AND_value
ori 1,1,b'000110'            * logically OR GPR 1 with RAG 1 OR_value
andi. 1,1,b'011111'          * logically AND GPR 1 with RAG 1 AND_value
rldimi 2,1,24,34             * rotate GPR 1 value and insert into GPR 2
                             * to align it with fields in TKM_PR register
```

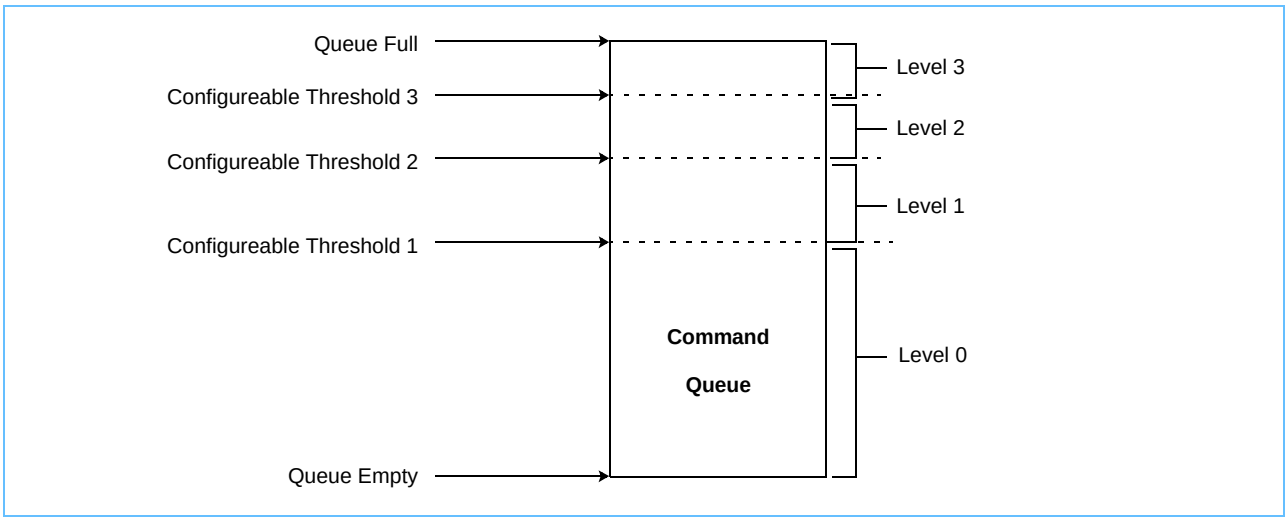
8.5.12 Feedback from Resources to Token Manager

The MIC and IOC (when configured for the IOIF protocol) provide feedback to the TKM about the state of their command queues, as shown in *Figure 8-1* on page 203. The MIC and IOC have registers to hold programmable thresholds. The feedback describes command-queue levels relative to these programmable thresholds, except that when an IOIF interface is configured for the BIF protocol, no such feedback is provided.

The command-queue levels, shown in *Figure 8-6*, are defined as follows, relative to the number of entries in the command queue:

- Level 0: number entries < Queue Threshold 1 field.
- Level 1: Queue Threshold 1 field ≤ number entries < Queue Threshold 2 field.
- Level 2: Queue Threshold 2 field ≤ number entries < Queue Threshold 3 field.
- Level 3: Queue Threshold 3 field ≤ number entries.

Figure 8-6. Threshold and Queue Levels



If `TKM_CR[FE]` enables the use of this queue-level feedback, the TKM grants tokens based on the queue-level feedback shown in *Table 8-4*. Ideally, feedback primarily limits the number of unallocated tokens granted and, secondarily, limits the sharing of unused tokens. If the feedback reaches Level 3, a `TKM_ISR` bit is set for the resource. If a `TKM_ISR` bit is '1' and the corresponding interrupt-enable bit in `TKM_CR` enables the interrupt, an External Interrupt condition is reported to the IIC. See *Section 9 PPE Interrupts* on page 237 for more information about IIC External Interrupt exceptions and how an external interrupt to the PPE can occur based on `TKM_ISR`.

Table 8-4. Token Allocation Actions Per Resource Queue Level

Token Allocation Actions	Queue Level for the Resource			
	0	1	2	3
Unallocated tokens are given out when available.	Yes	No	No	No
Unused tokens from one RAG are given to another RAG when the Allocation Decrementer is 0.	Yes	Yes	No	No
Tokens are generated for a RAG.	Yes	Yes	Yes	No

8.5.12.1 *MIC Feedback to Token Manager*

For each memory channel, the MIC has a command queue for reads and a command queue for writes. The MIC maintains counts for valid entries in these queues. These counts are compared to programmable threshold values in the MIC Token Manager Threshold Levels register (`MIC_TM_Threshold_n` [`n = '0','1'`]). The results of these comparisons are reported by the MIC to the TKM and used by the TKM as indicated in *Table 8-4*.

The highest-level information for any of these counts is reported by the MIC to the TKM. For example, if the command queue for reads has more entries than indicated by the value of the read-threshold-level 2 field in `MIC_TM_Threshold_n`, but fewer than the value of the read-threshold-level 3 field, and the command queue for writes has fewer entries than the value of the write-threshold-level 1 field, then the level reported to the TKM is level 2.

8.5.12.2 *IOC Feedback to Token Manager*

The IOC has logically separate command queues for each of the two IOIF buses, and for each bus there are separate queues for the commands received from the EIB and the IOIF. The commands remain in these queues until data is transferred on the IOIF bus. Thus, these queue entries represent the amount of outstanding data traffic for which a token has already been granted.

There is separate feedback to the TKM for each IOIF bus. For each IOIF bus, combined counts of the commands from the EIB and IOIF are used for the feedback, and the commands in the IOIF queue are only counted when a token has been obtained for their use of the IOIF bus. For commands in these queues, the IOC determines the number of commands that transfer data on the In bus and commands that transfer data on the Out bus. The IOIF0 counts are compared to programmable threshold values in the IOIF0 Queue Threshold register (`IOC_IOIF0_QueThshld`) and the IOIF1 counts are compared to programmable threshold values in the IOIF1 Queue Threshold register (`IOC_IOIF1_QueThshld`). The results of these comparisons are reported by the IOC to the TKM and used by the TKM as indicated in *Table 8-4*.

8.6 Configuration of PPE, SPEs, MIC, and IOC

8.6.1 Configuration Register Summary

Table 8-5 on page 229 shows the configuration registers or bits in the PPE, SPEs, MIC, and IOC units for controlling functions related to resource allocation. This table does not include all configuration registers that affect resource bandwidth. It only lists those related to resource allocation.

The Configuration Purpose column in *Table 8-5* shows the following functions that are controlled by these configuration registers:

- *Token-Request Enable or Override*—Configuration values shown in this row are reset to a state in which they do not require tokens. A configuration bit allows software to enable their dependency on obtaining tokens. Memory-mapped configuration registers holding these bits are in 4-KB pages with other registers that only need to be accessed by the hypervisor.
- *RAID Assignment*—The PPE and SPEs each have a configuration register that identifies the Resource Allocation Group ID (RAID) of the RAG associated with a resource access. The IOC has a configuration register that specifies the virtual-channel ID for outbound commands—that is, commands initiated by the PPE or an SPE. This virtual-channel ID could be associated with a RAID by the IOIF device, if that device implements Resource Allocation Management.
- *Address Map for Resources*—These registers provide a simplified view of the memory-map sufficient to recognize the addresses that access memory and IOIF1. The address ranges are defined either by MMIO registers accessible to software, or by non-mapped registers accessible only by the Configuration Ring bits at Power-On Reset (POR), as described in the *Cell Broadband Engine Hardware Initialization Guide*.

  If MMIO configuration registers are implemented, these registers are in 4-KB pages with other registers that only need to be accessed by the hypervisor (see *Section 11* on page 327). Addresses associated with IOIF0 or the BIF protocol are assumed to be addresses that do not correspond to one of the three local address spaces (memory, IOIF1, or MMIO registers). In a multi-CBE-processor environment, this allows all addresses associated with resources in or attached to the other CBE processor to be identified as using the BIF protocol.

  See *Section 8.6.2* on page 229 for details about the SPE mapping of addresses to resources. Like the SPE, the PPE has a similar coupling of IOIF1 address space to the CBE MMIO address space.
- *Multiple Tokens for Short Writes*—For memory-write operations shorter than 128 bytes, these configuration settings define whether multiple tokens are required for the PPE and IOC.The SPE always obtains two tokens for memory writes of 8 bytes or less.
- *Feedback Thresholds*—Configuration registers for the MIC and IOC that define the thresholds for reporting feedback to the TKM.

Table 8-5. Configuration Bits in the PPE, SPEs, IOC, and MIC

Purpose	Configuration Bits			
	PPE	SPEs	IOC	MIC
Token Request Enable or Override	BIU Configuration Ring:[1] • Resource Allocation Override. MMIO Register: • BIU Mode Setup 1 (BIU_ModeSetup1[1]).	SPE Configuration Ring:[1] • Resource Allocation Override. MMIO Register: • Resource Allocation Enable (RA_Enable[63]).	MMIO Register: • IOCmd Configuration (IOC_IOCmd_Cfg[17]).	Not applicable (the MIC is not a requester).
RAID Assignment	MMIO Register: • BIU Mode Setup 1 (BIU_ModeSetup1[2:3]).	MMIO Register: • RAID (RA_Group_ID[62:63]).	MMIO Register: • Virtual Channel For Outbound Accesses: IOCmd Configuration (IOC_IOCmd_Cfg[32:34] and IOC_IOCmd_Cfg[49:51]).	
Address Map for Resources	BIU Configuration Ring:[1] • MMIO Base Address. MMIO Registers: • for memory: BIU Mode Setup Register 1 (BIU_ModeSetup1[46:60] and BIU_ModeSetup1[14:28]). • for IOIF1: BIU Mode Setup Register 2 (BIU_ModeSetup2[22:31]).	SPE Configuration Ring:[1,2] • MC_BASE. • MC_COMP_EN. • IOIF1_COMP_EN. • BE_MMIO_Base.	Configuration Ring:[1] • BEI Base Address. • IOIF1 Base Address. • IOIF1 Base Address Mask. MMIO Registers: • Memory Base Address (IOC_MemBaseAddr). • IOC Base Address 1 (IOC_BaseAddr1). • IOC Base Address Mask 1 (IOC_BaseAddrMask1).	
Multiple Tokens for Short Writes	BIU Configuration Ring:[1] • Two-token decode for Non-Cacheable Unit (NCU) store.	Not applicable	MMIO Register: • IOCmd Configuration (IOC_IOCmd_Cfg[18] and IOC_IOCmd_Cfg[19]).	
Feedback Thresholds	Not applicable		MMIO Register: • IOIF0 Queue Threshold (IOC_IOIF0_QueThshld). • IOIF1 Queue Threshold (IOC_IOIF1_QueThshld).	MMIO Register: • Token Manager Threshold Levels n [n = 0,1] (MIC_TM_Threshold_n [n = 0,1]).

1. For details about the Configuration Ring bits and the Power-On Reset (POR) sequence, see the *Cell Broadband Engine Hardware Initialization Guide*.
2. See *Section 8.6.2* on page 229.

8.6.2 SPE Address-Range Checking

Each SPE has a defined real-address range for resources managed by the RAM facility. Before an SPE requests a token from the TKM, it checks the address range of the accessed resource. If the address falls into the SPE's defined range for a resource managed by the RAM facility, the SPE requests a token.

The range is defined in four registers that are written by the Configuration Ring bits. These registers, listed in *Table 8-6*, are not memory-mapped, but are only accessible by means of the Configuration Ring. The registers and the Configuration Ring bits are described in the *Cell Broadband Engine Hardware Initialization Guide*.

Table 8-6. SPE Address-Range Registers

Configuration-Ring Register	Description
BE_MMIO_Base	19-bit register that defines the BE_MMIO_Base starting address.
MC_BASE	15-bit register that defines the starting address of the MIC external memory.
MC_COMP_EN	15-bit register to specify the memory controller size. If (RA(0:14) and MC_COMP_EN) equal (MC_BASE and MC_COMP_EN), then hit MC range. See *Table 8-7*.
IOIF1_COMP_EN	10-bit register to specify the IOIF1 size.

The `MC_COMP_EN` register is assigned for system memory that belongs to the on-chip MIC. *Table 8-7* specifies the size of the `MC_COMP_EN`.

Table 8-7. MC_COMP_EN Setting

MC_COMP_EN	Memory Size
x‘4000’	2 TB
x‘6000’	1 TB
x‘7000’	512 GB
x‘7800’	256 GB
x‘7C00’	128 GB
x‘7E00’	64 GB
x‘7f00’	32 GB
x‘7f80’	16 GB
x‘7FC0’	8 GB
x‘7FE0’	4 GB
x‘7FF0’	2 GB
x‘7FF8’	1 GB
x‘7FFC’	512 MB
x‘7FFE’	256 MB
x‘7FFF’	128 MB

The IOIF1 range must start from the aligned address set by the `IOIF1_COMP_EN` register, and it is adjacent to `BE_MMIO_Base`. To guarantee that `BE_MMIO_Base` and the IOIF1 range do not touch each other, use the settings shown in *Table 8-8*. The IOIF0 range and IOIF1 range cannot be set independently. The IOIF0 range is the remaining address space (from the 42-bit RA) not defined by BE_MMIO_Base, IOIF1 range, and `MC_COMP_EN`.

Table 8-8. IOIF1_COMP_EN and BE_MMIO_Base Settings

IOIF1_COMP_EN	Size of IOIF1	BE_MMIO_Base	IOIF1 Start
1111111111	2 GB	be_mmio_base(10)= 0	be_mmio_base(0:9)|1 | 31'0
1111111110	4 GB	be_mmio_base(9) = 0	be_mmio_base(0:8)|1 | 32'0
1111111100	8 GB	be_mmio_base(8) = 0	be_mmio_base(0:7)|1 | 33'0
1111111000	16 GB	be_mmio_base(7) = 0	be_mmio_base(0:6)|1 | 34'0
1111110000	32 GB	be_mmio_base(6) = 0	be_mmio_base(0:5)|1 | 35'0
1111100000	64 GB	be_mmio_base(5) = 0	be_mmio_base(0:4)|1 | 36'0
1111000000	128 GB	be_mmio_base(4) = 0	be_mmio_base(0:3)|1 | 37'0
1110000000	256 GB	be_mmio_base(3) = 0	be_mmio_base(0:2)|1 | 38'0
1100000000	512 GB	be_mmio_base(2) = 0	be_mmio_base(0:1)|1 | 39'0
1000000000	1 TB	be_mmio_base(1) = 0	be_mmio_base(0)|1 | 40'0

8.7 Changing Resource-Management Registers with MMIO Stores

In some situations, it is not necessary to ensure the exact time at which a change to a resource-allocation facility has occurred and the side effects of changing such a facility occurs. If there is a need to synchronize such changes or limit the amount of variability of the side effects, PPE software can use the procedures described in this section.

8.7.1 Changes to the RAID

When a PPE context switch is to be made from one application to another application, or from one operating system to another operating system, it might be necessary to change the RAID used by the PPE or an SPE. (When an SPE context switch is to be made, use the procedures described in *Section 12 SPE Context Switching* on page 351.)

When a PPE context switch is made, the following sequence can be used to change the RAID:

1. Read the PPE's `BIU_ModeSetup1` register, and save this as part of the state.
2. After saving state, execute a **sync** instruction to ensure previous storage accesses are performed before the RAID change in the following step. (This **sync** is only useful if more precise use of the RAID is desired.)
3. Store the new RAID in the PPE's `BIU_ModeSetup1` register.
4. Execute a **sync** instruction to ensure the store to change the RAID is performed before subsequent storage accesses. (This **sync** is only useful if more precise use of the RAID is desired.)
5. Do memory accesses to restore the state of the next task.

This sequence assumes that the resource used up to the step in which the RAID is modified is attributed to the previous RAID, and the resource used after this step is attributed to the new RAID. This sequence does not guarantee that resource use is attributed to the correct RAG with 100% accuracy, because the PPE uses an implied token for each resource. Complete accuracy might be unimportant, because resource usage by the privileged software doing the context

switch and handling interrupts cannot be accurately attributed either. If complete accuracy is required, software can use the more lengthy sequence defined in *Section 8.7.6.1 Changes Involving Zero Allocation* on page 234.

8.7.2 Changing a Requestor's Token-Request Enable

Table 8-5 on page 229 shows the configuration registers that control whether a requester requires a token. To synchronize the side effects of a store to such a configuration register, software should subsequently load from the register and synchronize that load using a synchronization instruction such as a **sync**.

Before enabling token requests of a requester, software should configure the `TKM_CR`, `TKM_MBAR`, `TKM_IOIF0_AR`, and `TKM_IOIF1_AR` registers so that there are tokens granted to the RAG of the requester. Failure to do this could cause the requester to request a token that will not be granted, potentially resulting in a timeout and Checkstop.

8.7.2.1 *Changing PPE Token-Request Enable*

Software cannot turn off the PPE's token request after it is enabled. Software must only change `BIU_ModeSetup1[ResAllocEn]` when `BIU_ModeSetup1[ResAllocEn]` = '0'. If `BIU_ModeSetup1[ResAllocEn]` = '1', software must not change `BIU_ModeSetup1[ResAllocEn]`; otherwise, an outstanding storage access could timeout and a Checkstop could occur. To ensure that the PPE uses the new value for accesses, software should load from `BIU_ModeSetup1` after storing to it and software should synchronize that load using a synchronization instruction such as a **sync**.

8.7.2.2 *Changing SPE Token-Request Enable*

Software can change the `RA_Enable` register to enable SPE token requests at any time. Token-issue can be enabled by setting the Resource Allocation Enable bit, `RA_Enable[M]`, to '1' and the Resource Allocation Override bit, `RA_Enable[C]`, to '1'. The `RA_Enable[C]` field is read-only; its state can only be set by means of the Configuration-Ring load at POR, as described in the *Cell Broadband Engine Hardware Initialization Guide*.

Resource-allocation override is set by the Configuration Ring. `RA_Enable[M]` is cleared to '0' after POR. Software must set this bit to '1' to enable token-request. After `RA_Enable[M]` is set to '1', software should not reset this register unless all DMAC and atomic commands are complete. If there are any outstanding DMAC or atomic commands and `RA_Enable[M]` is '0', the behavior of the SPE is unpredictable.

Software must only change the `RA_Enable` register to disable SPE token requests when SPE accesses that require tokens are not occurring. To disable SPE token requests, PPE software can use the following sequence:

1. Perform an MMIO write of the `MFC_CNTL` register with the suspend DMA-queue (`Sc`) bit set to '1'.
2. Poll the `MFC_CNTL` register using an MMIO read until the MFC suspension requested in the previous step is complete.
3. Flush the contents of the MFC atomic unit to prevent a push from occurring while changing `RA_Enable`. The contents of the atomic unit can be flushed by writing a '1' to

MFC_Atomic_Flush[F] and then polling MFC_Atomic_Flush using an MMIO read until ‘0’ is returned.

4. Perform an MMIO write of the RA_Enable register to disable SPE token requests.
5. Perform an MMIO read of the RA_Enable register to ensure that the SPE uses the new value of RA_Enable for accesses.
6. To resume normal MFC operations, perform a MMIO write of the MFC_CNTL register with the restart MFC command (R) bit set to '1'.

8.7.2.3 *Changing I/O Token-Request Enable*

Software can change IOC_IOCmd_Cfg[TK] to ‘1’ at any time. However, this change only affects the token requests for new I/O accesses. I/O accesses that are queued in the IOC waiting for tokens are not affected by the change. To ensure that the IOC uses the new value for new reads and writes from the IOIF, software should load from IOC_IOCmd_Cfg after storing to it and synchronize that load using a synchronization instruction such as a **sync**.

Software must only change IOC_IOCmd_Cfg[TK] to ‘0’ when there are no concurrent or outstanding accesses from IOIF0 or IOIF1. Changing IOC_IOCmd_Cfg[TK] to ‘0’ when there is a read or write from an IOIF could cause the access to timeout because the read or write does not get the token that the IOC previously detected was needed.

8.7.3 Changing a Requestor’s Address Map

Table 8-5 on page 229 shows the configuration registers that controls the address map for resources. The CBE MMIO address map and the SPEs can only be configured by means of the Configuration Ring, so its configuration must be established at Power-On Reset.

To ensure that any side-effect of changing an IOC MMIO configuration register that controls the address map for resources has occurred, software should load from the last register modified and synchronize that load using a synchronization instruction such as a **sync** (the **sync** is needed after the read because these are pipelined registers). This change only affects the memory map for new I/O accesses. I/O accesses that are queued in the IOC waiting for tokens are not affected by the changing address map. Because the address map for IOIF1 is controlled by two registers, it is not possible to change both at the same instant. As a result, software should only change the IOC_BaseAddr1 register or the IOC_BaseAddrMask1 register when either IOC_IOCmd_Cfg[TK] is ‘0’ or there are no concurrent or outstanding accesses to IOIF1 from IOIF0 while the IOC_BaseAddr1 register or IOC_BaseAddr1 register is being modified.

Changing the PPE address map for resources might require changing both the BIU_ModeSetup1 register and the BIU_ModeSetup2 register. After changing one of these registers, software must avoid having a temporary memory map that has a single address mapped to two resources. To ensure that any side-effect of changing either the BIU_ModeSetup1 register or the BIU_ModeSetup2 register has occurred, software should load from the modified register after storing to it and synchronize that load using a synchronization instruction such as a **sync**. If modifying both of these registers, software only needs to load from the register modified last and then synchronize that load.

8.7.4 Changing a Requestor's Use of Multiple Tokens per Access

For some period of time after performing a store to change `IOC_IOCmd_Cfg[RMW:SXT]`, the IOC token requests could be based on either the new or old value of `IOC_IOCmd_Cfg[RMW:SXT]`. If software needs to ensure the new value is used, software can load from the modified register and synchronize that load using a synchronization instruction such as a **sync**. This change only affects the token requests for new I/O accesses. I/O accesses that are queued in the IOC waiting for tokens are not affected by the change.

8.7.5 Changing Feedback to the TKM

After a store to `MIC_TM_Threshold_n [n = '0','1']`, `IOC_IOIF0_QueThshld`, or `IOC_IOIF1_QueThshld`, to ensure that the new values for these registers are used for feedback to the TKM, software can load from a modified register and synchronize that load using a synchronization instruction such as a **sync**.

8.7.6 Changing TKM Registers

To ensure that the new register values are used after a store to the `TKM_CR`, `TKM_PR`, or `TKM_PMCR` registers, software can load from a modified register and synchronize that load using a synchronization instruction such as a **sync**.

After a store to the `TKM_MBAR`, `TKM_IOIF0_AR`, or `TKM_IOIF1_AR` registers, the TKM uses the new allocation rate after the corresponding RAG Rate Decrementer expires. To ensure that the new values for these registers are used, software can load from a modified register, synchronize that load using a synchronization instruction such as a **sync**.

8.7.6.1 *Changes Involving Zero Allocation*

Before changing the `TKM_CR`, `TKM_MBAR`, `TKM_IOIF0_AR`, or `TKM_IOIF1_AR` registers so that a RAG is no longer allocated some resource, software must ensure that there are no outstanding token requests for that resource from that RAG and that there will be no such future requests. Requesters in such a RAG for such a resource are not granted tokens and might time-out, potentially resulting in a Checkstop.

Software can ensure that this is the case for SPEs by using the sequences in the *Section 12 SPE Context Switching* on page 351. For some value `r`, if RAG `r` is no longer going to be allocated some resource, then during the restore sequence software would load a value other than `r` into the RAID field of the SPE `RA_Group_ID` register.

Ensuring that the PPE has no outstanding request for RAG `r` is more complicated, because the PPE uses an implied token for each resource. Ensuring that previous storage accesses are performed does not guarantee that replacement tokens have been obtained. The following sequence should be executed from the PPE before changing the `TKM_CR`, `TKM_MBAR`, `TKM_IOIF0_AR`, or `TKM_IOIF1_AR` registers to eliminate a resource allocation for RAG `r`:

1. Store the new RAID, where this RAID is not `r`, in the PPE's `BIU_ModeSetup1` register.
2. Read the `BIU_ModeSetup1` register.
3. Execute a **sync** instruction to ensure the read has been performed before subsequent storage accesses.

4. If RAG `r` is no longer allocated a memory resource, then, for each memory bank, execute a **dcbf** instruction and a load instruction where both instructions use the same operand address corresponding to the memory bank. Software controls whether a RAG is allocated tokens for a memory resource. For example, setting any of the `TKM_CR[Ei]` bits[44:47] to '0' is one way to stop granting tokens to the corresponding RAG.
5. If RAG `r` is no longer allocated an IOIF Out data bus resource, then store to a location attached to the IOIF.
6. If RAG `r` is no longer allocated an IOIF In data bus resource, then load from a location attached to the IOIF.
7. Execute a **sync** instruction to ensure that accesses in the preceding steps have been performed.

8.8 Latency Between Token Requests and Token Grants

Enabling Resource Allocation Management adds latency because of the time required to request and grant tokens. If a token is already available and there are no other requests in the same RAG for the same resource, there is a 6 `NClk`/2 cycle latency in the TKM from request to grant. If the token is not already available, there might be additional latency due to the wait time in the TKM request queue. There is also latency in each requester, delay in transferring the request from the requester to the TKM, and delay in transferring the grant from the TKM to the requester.

8.9 Hypervisor Interfaces

The CBE processor's hypervisor facility is described in *Section 11 Logical Partitions and a Hypervisor* on page 327. Hypervisor software may find it useful to define a few hypervisor function calls that manage the relationship between the RAM facility's RAG allocations and the hypervisor's logical-partitioning facility. This may be useful, for example, when quality of service (QoS) must be guaranteed for specific resources used by a specific logical partition.

For example, hypervisor function calls might do the following:

- *Acquire a RAG*—This function would acquire a RAG for a logical partition. Function arguments might include the logical-partition ID, minimum acceptable memory-access allocation, and minimum acceptable I/O-access allocation. Return values might include the allocated RAID (or more detailed RAG descriptor) and various success or failure indicators. All resource requestors belonging to that logical partition would become a member of the new RAG. When associating a RAG with a logical partition that is currently able to run, the partition should be immediately removed from an unallocated RAG and placed into the new RAG. The acquisition of a new RAG in this manner would require some initial allocation for memory and I/O resources, or the partition would not make forward progress.
- *Allocate a RAG*—This function would assign a portion of resource's time that a RAG can use the resource. Function arguments might include the RAID or descriptor of the RAG that was acquired by calling the *Acquire a RAG* function, the resource to be allocated, and the portion of the resource's time to allocate. Return values might include the remaining portion of the resource's time, after successful allocation, and various success or failure indicators.
- *Release a RAG*—This function would release a RAG associated with a logical partition. Function arguments would include the RAID or descriptor of the RAG to release. Return val-

ues might include various success or failure indicators. Any requestor owned by the logical partition associated with this RAG would have its resources allocated by the Unallocated RAG.

See *Section 11.5.1 Combining Logical Partitions with Resource Allocation* on page 346 for a short overview of how these two facilities can be used together.

9. PPE Interrupts

This chapter describes all interrupts that are handled by the PPE, or by an external interrupt controller. Additional interrupts caused by SPE Events, such as signal-notification events, are handled locally by each SPE as described in *Section 18.3 SPU Interrupt Facility* on page 468.

9.1 Introduction

Exceptions and interrupts are closely related. The *PowerPC Architecture* and the *PowerPC Compiler Writer's Guide* define an *exception* as an error, external event, or other condition requiring attention, such as an instruction that cannot be executed, the assertion of an external signal, or a timer event. An exception can set status information in a register, indicating the nature of the exception, and it can cause a corresponding *interrupt,* if a corresponding interrupt is enabled and (if applicable) unmasked. It is possible for a single instruction to cause multiple exceptions and for multiple exceptions to cause a single interrupt. By masking interrupts, software can also support polling as an alternative to interrupts.

An interrupt is handled by an *interrupt handler* (sometimes called an *interrupt service routine*, *error handler*, or an *exception handler*)—a privileged program that determines the nature of the exception and decides how to respond. Interrupts are thus the mechanisms by which a processor identifies the exception and passes control to the interrupt handler.

Because they are closely related, the terms *exception* and *interrupt* are sometimes used interchangeably—either one meaning what the other was originally intended to mean in the *PowerPC Architecture*. For example, the *PowerPC Microprocessor Family: The Programming Environments for 64- and 32-Bit Microprocessors* document uses the term *exception* to mean what the *PowerPC Architecture* distinguishes as *exception* and *interrupt*. That PEM document also uses the term *interrupt* to mean an *asynchronous exception*. The original meanings, which are used in this chapter, are that one term (*exception*) names the cause of an event, and the other term (*interrupt*) names the response to that event.

In the CBE processor, exceptions can arise due to the execution of PPE instructions or from the occurrence of external events, including the execution of SPU instructions or MFC commands or from an I/O or memory device. All interrupts, except those resulting from SPE events (*Section 18 SPE Events* on page 463), are directed to and handled by the PPE or an external interrupt controller.

The CBE processor provides facilities for:

- Routing interrupts and interrupt-status information to the PPE or to an external interrupt controller attached to one of the two I/O interfaces.
- Prioritizing interrupts presented to the PPE or to an external interrupt controller.
- Generating interprocessor interrupts between one PPE thread and another.

The PPE is supported by two Internal Interrupt Controllers (IICs), one for each PPE thread. The IICs generate, route, and maintain interrupt information for the PPE threads. Each SPE has a set of interrupt registers for masking interrupt conditions, holding the status of interrupt conditions, and routing interrupts.

The *PowerPC Operating Environment Architecture, Book III* and the *PowerPC Microprocessor Family: The Programming Environments for 64- and 32-Bit Microprocessors* documents describe PowerPC exception ordering, interrupt definitions, interrupt priorities, interrupt synchronization, and interrupt processing. Similar details regarding CBE-specific exceptions and interrupts can be found in the *Cell Broadband Engine Architecture* document.

9.2 Summary of Interrupt Architecture

The PowerPC Architecture defines two classes of interrupts:

- *Instruction-Caused*—Interrupts caused by the execution of a PowerPC instruction, and thus synchronous with the PowerPC instruction pipeline. In the CBE processor, these include the execution of all PPE instructions.
- *System-Caused*—Interrupts caused by an event external to the PowerPC instruction pipeline, or otherwise not synchronous with the execution of PowerPC instructions. From the viewpoint of the PPE, interrupts caused by the execution of an SPU instruction or an MFC command are system-caused.

In addition, the PowerPC Architecture defines two kinds of interrupts caused by instructions:

- *Precise*—These are interrupts in which the architecturally visible processor state (in the CBE processor, this is the PPE state) is known at the time of the exception that caused the interrupt. After a precise interrupt is handled, execution can be resumed where it left off. A precise interrupt is caused by an exception that was generated when the instruction was fetched or executed.
- *Imprecise*—These are interrupts in which the architecturally visible processor state (PPE state) is not guaranteed to be known at the time of the exception that caused the interrupt.

The PPE exception and interrupt architecture is consistent with these PowerPC Architecture definitions. In the CBE processor, instruction-caused (synchronous) interrupts are precise with respect to the PPE[19], and system-caused (asynchronous) interrupts are imprecise with respect to the PPE[20].

Table 9-1 on page 239 summarizes the characteristics of interrupts supported by the PPE:

- *Precise*—As defined above. For consistency across all interrupts, this parameter is identified in *Table 9-1* for both instruction-caused and system-caused interrupts.
- *Maskable*—With an interrupt-mask bit and a corresponding exception-status bit that can be read by polling, as an alternative to taking an interrupt when an exception occurs. Software can delay the generation of these interrupts by masking them.
- *Can Be Disabled*—Prevented from generating an exception or a corresponding interrupt.
- *Context-Synchronizing*—Previously issued instructions are completed before continuing with program execution. Context-synchronizing instructions empty (flush) the instruction-prefetch queue and start instruction fetching in the context established after all preceding instructions have completed execution. See *Section 20.1.2.4* on page 556 for details about context-synchronization of PPE interrupts.

---

19. In the *PowerPC Architecture*, certain floating-point exceptions can be imprecise, based on mode settings, but the PPE does not conform to the architecture in this regard.
20. But see *Section 9.5.2* on page 246 for details about the precise form of the Machine Check interrupt.

In *Table 9-1*, the ability to mask and disable interrupts assumes that the underlying function that could cause such an interrupt is enabled. For example, Floating-Point Unavailable interrupts can be prevented by disabling the underlying function of executing any floating-point instructions (`MSR[FP]` = ‘0’); however, *Table 9-1* assumes that these functions are enabled.

Table 9-1. PPE Interrupts

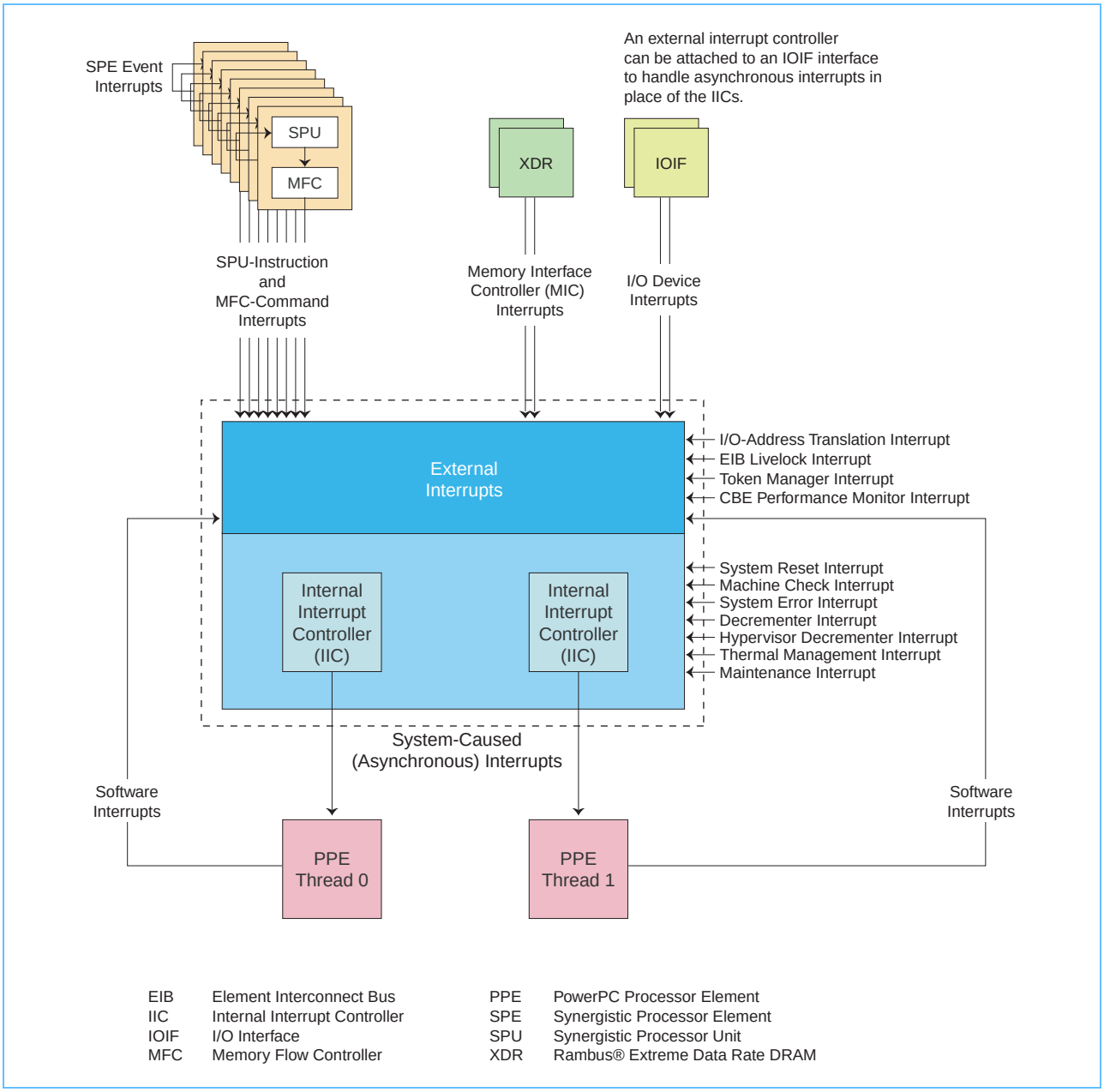
Interrupt Class	Interrupt		Precise	Can Be Masked	Can Be Disabled	Context Synchronizing	Reference
Instruction-Caused (Synchronous)	Data Storage		Yes	No	No	Yes	*Section 9.5.3* on page 248
	Data Segment						*Section 9.5.4* on page 249
	Instruction Storage						*Section 9.5.5* on page 250
	Instruction Segment						*Section 9.5.6* on page 251
	Alignment						*Section 9.5.8* on page 251
	Program						*Section 9.5.9* on page 252
	Floating-Point Unavailable						*Section 9.5.10* on page 254
	System Call						*Section 9.5.13* on page 255
	Trace				Yes		*Section 9.5.14* on page 256
	VXU Unavailable						*Section 9.5.15* on page 256
	Maintenance (instruction-caused)			Yes			*Section 9.5.17* on page 258
System-Caused (Asynchronous)	System Reset		No[1]	No	No[2]	Yes	*Section 9.5.1* on page 245
	Machine Check		Yes[3]		Yes	Yes[3]	*Section 9.5.2* on page 246
	System Error		No[1]	Yes		Yes	*Section 9.5.16* on page 257
	Decrementer				Yes[4]		*Section 9.5.11* on page 254
	Hypervisor Decrementer			No	Yes		*Section 9.5.12* on page 254
	Thermal Management			Yes			*Section 9.5.18* on page 260
	Maintenance (system-caused)[5]			No			*Section 9.5.17* on page 258
	External (direct and mediated)[6]	SPU Instructions		Yes[7,8]	Yes[4]		*Section 9.6.3* on page 268
		MFC Commands					
		Memory Interface Controller (MIC) Auxiliary Trace Buffer Full		Yes			*Section 9.6.4.2* on page 270
		I/O devices (IOIF0, IOIF1)		Yes[8,9]	Yes[4,8]		*Section 9.6.4.3* on page 270
		I/O-Address Translation		Yes[8]	Yes[4]		*Section 9.6.4.4* on page 271
		Element Interconnect Bus (EIB) Possible Livelock Detection					*Section 9.6.4.5* on page 272
		Token Manager					*Section 9.6.4.6* on page 273
		CBE Performance Monitor		Yes[7,8]			*Section 9.6.4.7* on page 273
		Software Interrupt[10]		Yes[8]			*Section 9.6.2.4* on page 268

1. These interrupts are recoverable.
2. But one of the several sources for this interrupt may be disabled.
3. See *Section 9.5.2* on page 246.
4. The interrupt can be disabled with `MSR[EE]`, but it cannot be *independently* disabled without also disabling all External interrupts.
5. A system-caused Maintenance interrupt cannot occur without an appropriate boundary-scan debug tool.
6. See *Section 9.5.7* on page 251 for the difference between Direct and Mediated External Interrupts.
7. By means of specific mask and status registers.
8. By the minimum-priority mechanism of the `IIC_CPL0` and `IIC_CPL1` registers (*Section 9.6.2* on page 263).

9. A SouthBridge (I/O interface chip) may support enabling and masking.
10. An interrupt to one of the PPE threads using an Interrupt Generation Port (IGP), `IIC_IGP0` or `IIC_IGP1`. Also called an InterProcessor Interrupt (IPI). There is no corresponding exception, and software can disable such interrupts by simply not issuing them.

Figure 9-1 on page 240 illustrates the organization of interrupt handling, assuming that the PPE (rather than an external interrupt controller attached to an I/O interface) handles interrupts. The CBE processor has two Internal Interrupt Controllers (IICs), one for each PPE thread. These controllers service all system-caused (asynchronous) interrupts, including External interrupts.

Figure 9-1. Organization of Interrupt Handling



The only interrupts shown in *Figure 9-1* that are not handled by the PPE are those arising from local SPE events. These are shown as “SPE Event Interrupts” in *Figure 9-1*, and they are handled locally by each SPE as described in *Section 18.3 SPU Interrupt Facility* on page 468.

9.3 Interrupt Registers

Table 9-29 summarizes the interrupt-related registers in the PPE. The interrupt registers allow privileged software on the PPE to select which PPE exceptions are allowed to generate an interrupt to the PPE. For details, see the *PowerPC Operating Environment Architecture, Book III,* the *PowerPC Microprocessor Family: The Programming Environments for 64- and 32-Bit Microprocessors* document, and the *Cell Broadband Engine Registers* document.

Table 9-2. PPE Interrupt Register Summary (Page 1 of 2)

Register Name	Mnemonic	Description	Read/ Write
Thread-Dependent Registers[1] (*PowerPC Architected Registers*)			
Machine State Register	MSR	Defines the state of the processor. When an exception occurs, the contents of MSR are saved in SRR1, and a new set of bits are loaded into MSR as determined by the exception.	R/W
Machine Status Save/Restore Register 0	SRR0	For non-hypervisor interrupts, records the effective address (EA) at which an interrupt occurs, and provides the return address to the interrupted program when an **rfid** (or **rfi**) instruction is executed. It also holds the EA for the instruction that follows a System Call (**sc**) instruction.	R/W
Machine Status Save/Restore Register 1	SRR1	For non-hypervisor interrupts, records the exception status and the contents of the MSR when an **rfid** (or **rfi**) instruction is executed.	R/W
Hypervisor Machine Status Save/Restore Register 0	HSRR0	For hypervisor interrupts, records the effective address (EA) at which an interrupt occurs, and provides the return address to the interrupted program when an **hrfid** instruction is executed.	R/W
Hypervisor Machine Status Save/Restore Register 1	HSRR1	For hypervisor interrupts, records the exception status and the contents of the MSR when an **hrfid** instruction is executed.	R/W
Floating-Point Status and Control Register	FPSCR	Enables floating-point interrupts. Records exceptions generated by floating-point operations and the type of result produced by a floating-point operation.	R/W
Data Storage Interrupt Status Register	DSISR	Records the cause of DSI and alignment exceptions.	R/W
Decrementer	DEC	Causes a decrementer exception after a programmable delay.	R/W
Data Address Register	DAR	Records the effective address generated by a memory- access instruction if the access causes an exception (for example, an alignment exception).	R/W
Data Address Breakpoint Register	DABR DABRX	Specifies an effective address (EA) for load and store accesses to generate a data-address breakpoint interrupt.	R/W

Table 9-2. PPE Interrupt Register Summary (Page 2 of 2)

Register Name	Mnemonic	Description	Read/ Write
Address Compare Control Register	ACCR	Specifies a Data Address Compare match for Instruction Storage interrupts.	R/W
Thread-Dependent Registers[1] *(Implementation-specific Registers)*			
Logical Partition Control Register	LPCR	Controls logical (hypervisor) partitioning. Enables Mediated External exceptions. Controls whether External Interrupts set MSR[HV] to '1' or leave it unchanged. The LPCR is a partially shared between the two PPE threads.	R/W
Thread Status Register Local	TSRL	Specifies thread priority, and reports forward-progress timer. When a thread reads its own TSR register, this register is called the Thread Status Register Local (TSRL). When a thread reads the TSR for the other thread, this register is called the Thread Status Register Remote (TSRR).	R/W
Thread Status Register Remote	TSRR		
Thread-Independent Registers[2] ***(PowerPC Architected Registers)***			
Hypervisor Decrementer	HDEC	Provides a means for the hypervisor to manage timing functions independently of the Decrementer, which is managed by virtual partitions.	R/W
Control Register	CTRL	Control and status for threads.	R/W
Thread-Independent Registers[2] *(Implementation-specific Registers)*			
Hardware Implementation Dependent Register 0	HID0	Enables thermal management interrupt, system error interrupt, extended external interrupt, precise or imprecise form of the Machine Check interrupt, and attention instruction.	R/W
Hardware Implementation Dependent Register 1	HID1	Enables configuration ring system reset interrupt address register. Forces instruction-cache parity error. Controls trace bus.	R/W
Thread Switch Control Register	TSCR	Enables Decrementer wakeup, priority boost for system-caused interrupts.	R/W
Thread Switch Time-Out Register	TTR	Specifies thread time-out flush values.	R/W

1. These registers are duplicated, one for each PPE thread.
2. These registers serve interrupt-handling for both PPE threads.

9.4 Interrupt Handling

PPE interrupts are handled according to the rules defined in *PowerPC Operating Environment Architecture, Book III*. The interrupts are handled in either privileged (supervisor) or hypervisor mode, depending on the interrupt. In this section, the phrase *the interrupt is taken* refers to the *PowerPC Architecture* interrupt facility.

Upon entering the privileged state, a small portion of the processor's current state is saved in the Machine Status Save/Restore Registers (SRR0, SRR1, HSRR0, and HSRR1). The Machine State register (MSR) is updated, and instruction fetch and execution resumes at the real address associated with the interrupt (called the *interrupt vector*). Since the Machine Status Save/Restore regis-

ters are serially reused by the PPE, all interrupts except System Reset and Machine Check are ordered as defined in the *PowerPC Architecture*. The save/restore registers are described in the *Cell Broadband Engine Registers* specification.

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

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

Each PPE thread is viewed as an independent processor complete with separate exceptions and interrupt handling. Exceptions can occur simultaneously for each thread. The PPE supports concurrent handling of interrupts on both threads by duplicating some registers defined by the PowerPC Architecture. The registers associated with Interrupt handling are summarized in *Table 9-2* on page 241. See *Table 9-38* on page 287 for the tread targets of various interrupt types.

See *Section 9.6 Direct External Interrupts* on page 262 for further details about handling External Interrupts.

For details about PowerPC interrupt handling, see the *PowerPC Operating Environment Architecture, Book III* and the *PowerPC Microprocessor Family: The Programming Environments for 64- and 32-Bit Microprocessors* documents. For details about CBEA-specific interrupt handling, see the *Cell Broadband Engine Architecture*.

9.5 Interrupt Vectors and Definitions

Table 9-1 on page 239 lists the types of interrupts caused by PPE instructions and system conditions. *Table 9-3* provides references to interrupt vectors and the detailed subsections, below, that describe the PPE interrupt causes and behaviors.

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

Further descriptions of the PowerPC interrupts and their causes are given in the PowerPC Architecture. *Book I and Book III* and the *PowerPC Microprocessor Family: The Programming Environments for 64- and 32-Bit Microprocessors* document. For a list identifying PowerPC and CBE interrupt differences, see *Section 9.17 Relationship Between CBE and PowerPC Interrupts* on page 292.

Table 9-3. Interrupt Vector and Exception Conditions

Interrupt Type	Interrupt Vector Effective Address (EA)	Causing Conditions (Exceptions)
System Reset	Selectable, based on `HID1[dis_sysrst_reg]`	See *Section 9.5.1* on page 245.
Machine Check	x‘00..00000200’	See *Section 9.5.2* on page 246.
Data Storage	x‘00..00000300’	See *Section 9.5.3* on page 248.
Data Segment	x‘00..00000380’	See *Section 9.5.4* on page 249.
Instruction Storage	x‘00..00000400’	See *Section 9.5.5* on page 250.
Instruction Segment	x‘00..00000480’	See *Section 9.5.6* on page 251.
External	x‘00..00000500’	See *Section 9.5.7* on page 251.
Alignment	x‘00..00000600’	See *Section 9.5.8* on page 251.
Program	x‘00..00000700’	See *Section 9.5.9* on page 252.
Floating-Point Unavailable	x‘00..00000800’	See *Section 9.5.10* on page 254.
Decrementer	x‘00..00000900’	See *Section 9.5.11* on page 254.
Hypervisor Decrementer	x‘00..00000980’	See *Section 9.5.12* on page 254.
System Call	x‘00..00000C00’	See *Section 9.5.13* on page 255.
Trace	x‘00..00000D00’	See *Section 9.5.14* on page 256.
Performance Monitor	x‘00..00000F00’	This exception is optional in the *PowerPC Architecture*. It is not implemented by the PPE.
VXU Unavailable	x‘00..00000F20’	See *Section 9.5.15* on page 256.
System Error	x‘00..00001200’	See *Section 9.5.16* on page 257. This exception is not defined in the *PowerPC Architecture*; it is specific to the processor implementation.
Maintenance	x‘00..00001600’	See *Section 9.5.17* on page 258. This exception is not defined in the *PowerPC Architecture*; it is specific to the processor implementation.
Thermal Management	x‘00..00001800’	See *Section 9.5.18* on page 260. This exception is not defined in the *PowerPC Architecture*; it is specific to the processor implementation.
Note: Interrupt Vectors not listed in this table are reserved.		

9.5.1 System Reset Interrupt (Selectable or x‘00..00000100’)

The System Reset interrupt can be used for Power-On Reset (POR), to restart a running processor, and to resume a suspended thread. A System Reset interrupt is caused by the assertion of one of the System Reset interrupt signals from the internal logic or if an exception occurs for a suspended thread.

On a System Reset Interrupt, the PPE starts thread 0 from the address specified in the *PPE SReset Vector* field of the Configuration Ring (see the *Cell Broadband Engine Hardware Initialization Guide*). As a result of the System Reset Interrupt, the Hypervisor and 64-Bit-Mode bits, `MSR[HV]` and `MSR[SF]`, are both set to ‘1’, so that the PPE comes up in hypervisor mode.

A System Reset interrupt is generated for a suspended thread if a Thermal Management, System Error, External, or Decrementer interrupt exception exists, or if a thread is enabled by a write to the Control Register (`CTRL`). The System Reset interrupt vector’s real address for thread 0 is selected based on the setting of `HID1[dis_sysrst_reg]`. If `HID1[dis_sysrst_reg]` is set to ‘1’, the interrupt vector is x‘000..0100’. If `HID1[dis_sysrst_reg]` = ‘0’, bits [22:61] of the interrupt vector are defined by the configuration ring and bits [0:21] and bits [62:63] are set to ‘0’. The interrupt vector for thread 1 is fixed at x‘000..0100’. For details on configuration-ring settings, see *Cell Broadband Engine Hardware Initialization Guide*.

When a thread is resumed, the PPE sets the corresponding Thread Enable bit in the `CTRL` register to ‘1’. Resuming a thread due to External and Decrementer interrupts is maskable by `TSCR[WEXT]` plus `TSCR[WDEC0]` for thread 0 or `TSCR[WDEC1]` for thread 1. Resuming a thread due to a System Error or Thermal Management interrupt is maskable by `HID0[syserr_wakeup]` and `HID0[therm_wakeup]`, respectively.

The PPE treats the System Reset interrupt conditions as a context-synchronizing operation as defined in the PowerPC Architecture. When the PPE takes a System Reset interrupt, the thread’s priority is set to High, the register state is altered as defined in *Table 9-4* on page 246, and instruction fetch and execution resume at the effective address specified in *Table 9-3* on page 244.

Table 9-4. Registers Altered by a System Reset Interrupt

Register	Bits	Setting Description
Thread-Dependent Registers		
SRR0	0:63	Set to the effective address of the instruction that the processor would have attempted to execute next if no exception condition were present.
SRR1	0:41 45:63	These bits of SRR1 are set as defined in the PowerPC Architecture. SRR1[62] is always a copy of the MSR[RI] bit at the time of the interrupt.
	42:44	System Reset Reason Code 000 Reserved 001 Reserved 010 System Reset due to a Thermal Management interrupt to a suspended thread (HID0[therm_wakeup] must be set to ‘1’) 011 System Reset due to a Decrementer interrupt to a suspended thread (TSCR[WDEC0] for thread0 or TSCR[WDEC1] for thread1 must be set to ‘1’) 100 System Reset due to an External Interrupt to a suspended thread (TSCR[WEXT] must be set to ‘1’) 101 Thread resumed due to 1) a System Reset due to a power-on reset (POR) or 2) a write to the Control Register (CTRL) with the thread enable bit set. Software can check HID0 and HID1 to see if they are set to all ‘0’s to determine if condition 1 occurred. If HID0 and HID1 are not all ‘0’s, then condition 2 occurred. 110 Thread resumed due to a System Error (HID0[syserr_wakeup] must be set to ‘1’) 111 Reserved
MSR	0:63	See *Table 9-40 Machine State Register (MSR) Bit-Settings Due to Interrupts* on page 290.
TSRL	11:12	11 High thread priority
Thread-Independent Register		
`CTRL`	8:9	Bit 8 is the thread enable for thread0, and bit 9 is the thread enable for thread1. Depending on which thread took the interrupt, the corresponding thread enable bit will be set.

9.5.2 Machine Check Interrupt (x‘00..00000200’)

A Machine Check interrupt occurs when no higher-priority interrupts exist and a Machine Check condition occurs. In the PPE, there is one possible source for a Machine Check interrupt—a data error on a caching-inhibited load (such as a nonexistent memory address). If two Machine Checks occur too close together, a Checkstop can occur (if there is not enough time to bring down a partition). A Checkstop is a full-stop of the processor that requires a System Reset to recover. If either thread has Machine Checks disabled (`MSR[ME]` = ‘0’) and a Machine Check occurs, the PPE also enters a checkstop condition. The PPE does not modify the DSISR or DAR when a Machine Check interrupt is taken.

The *PowerPC Architecture* defines the Machine Check interrupt as a system-caused interrupt. in the CBE implementation:

- If `HID0[en_prec_mchk]` = ‘1’ (“precise”), only the thread that causes a Machine Check interrupt—that is, thread issuing the offending caching-inhibited load instruction—will take the interrupt, and the interrupt will appear to be instruction-caused (synchronous) from the viewpoint of that thread (at the expense of PPE performance), and the interrupt will be context-synchronizing, and the interrupt will be recoverable.

- If `HID0[en_prec_mchk]` = ‘0’ (“imprecise”), both threads will take a Machine Check interrupt, and the interrupt will appear to be system-caused (asynchronous) to both threads, and the interrupt will not be context-synchronizing, and the interrupt will not be recoverable.

However, the interrupt—in either its precise or imprecise form—is system-dependent in the sense that it reflects the configuration of the system; thus, the same code sequence on a different system may not cause a Machine Check interrupt.

The advantage of a precise Machine Check is that a software-defined interrupt handler can potentially recover from the interrupt with predictable results. An imprecise Machine Check has the advantage of yielding better hardware performance (for caching-inhibited loads) with the risk of nonrecoverability if a Machine Check occurs. Precise Machine Check interrupts are enabled depending on the setting of `HID0[en_prec_mchk]`. If `HID0[en_prec_mchk]` = ‘0’, then the processor is in normal operation and Machine Checks are imprecise and have the potential to cause a checkstop if multiple loads outstanding can cause a machine check. Otherwise, if `HID0[en_prec_mchk]` = ‘1’, then the processor will flush all instructions after any caching-inhibited load is issued and block them at dispatch until the load completes so that Machine Checks can be precise.

Software running in imprecise mode that depends on the return of a data error (DERR) on a caching-inhibited load—and thus the imprecise Machine Check—to indicate the existence of a device should perform the load with only a single thread active. If the load is performed while both threads are active, it is possible to receive the interrupt when the other thread is in a nonrecoverable state resulting in a checkstop condition. In precise mode, software can run with both threads enabled for the purpose of polling devices that return a DERR.

The PPE treats the precise Machine Check interrupt condition as a context-synchronizing operation as defined in the PowerPC Architecture. Imprecise Machine Checks are not context synchronizing. To guarantee the exception when Machine Checks are imprecise, a **sync** (L=0) instruction is needed after the caching-inhibited load.

When a condition exists for an imprecise Machine Check, both threads of the PPE, if active, take the imprecise Machine Check interrupt. For the thread that issues the imprecise Machine-Check-causing load, the interrupt appears to be instruction-caused; on the other thread, the interrupt is system-caused.

When the PPE takes a Machine Check interrupt, the register state is altered as defined in *Table 9-5*, and instruction fetch and execution resume at the effective address specified in *Table 9-3* on page 244.

Table 9-5. Registers Altered by a Machine Check Interrupt

Register	Bits	Setting Description
SRR0	0:63	Set to the effective address of the instruction that the processor would have attempted to execute next if no exception condition were present.
SRR1	0:63	All bits cleared to ‘0’ except those set by the MSR. SRR1 Specifics: [33:36] = 0 [42:47] = 0 [0:32], [37:41], [48:61], [63] = MSR values. SRR1[62] is set to ‘0’ for an imprecise Machine Check. For a precise Machine Checks, SRR1[62] is set to the MSR value.
MSR	0:63	See *Table 9-40 Machine State Register (MSR) Bit-Settings Due to Interrupts* on page 290.

9.5.3 Data Storage Interrupt (x‘00..00000300’)

A Data Storage interrupt occurs when no higher-priority interrupts exists and an exception condition related to a data access occurs. Data Storage interrupts are precise and are not maskable. The Data Storage interrupt is implemented as defined in the PowerPC Architecture. The following conditions cause a Data Storage exception:

- When any byte of the storage location specified by a load, store, **icbi**, **dcbz**, **dcbst**, or **dcbf** instruction cannot be translated from a virtual address to a real address. Data translation must be enabled for this exception to occur (`MSR[DR]` set to ‘1’).
- When the data access violates the storage protection as defined in the PowerPC Architecture. The storage protection is based on the [`Ks`] and [`Kp`] bits in the Segment Lookaside Buffer (SLB), and the [PP] bits in the page table entry (PTE) when `MSR[DR]` is set to ‘1’. When `MSR[DR]` is set to ‘0’, a data storage protection violation occurs if `LPCR[LPES]` bit 1 = ‘0’ and `MSR[HV]` = ‘0’ or if `LPCR[LPES]` bit 1 = ‘1’ and the data effective address is greater than or equal to the `LPCR[RMLS]`.
- When a data address compare or data address breakpoint match occurs.

The PowerPC Architecture describes the processor behavior for the following situation as implementation-specific: a Data Storage interrupt is generated because an **stwcx.** or **stdcx.** instruction is executed, storage is not updated, and there is a Data Storage exception. For this case, the PPE produces a Data Storage interrupt, regardless of the condition for update of storage.

A Data Storage interrupt in the PPE can occur at two points during the translation of a virtual address to a real address, depending on the Translation Lookaside Buffer (TLB) management mode. For software TLB management, the Data Storage interrupt occurs if the translation is not found in the TLB. Software TLB management is controlled by the `LPCR[TL]` bit. For hardware TLB management, the interrupt occurs after an unsuccessful search of the page table.

The PPE treats the Data Storage interrupt conditions as a context-synchronizing operation as defined in the PowerPC Architecture. When the PPE takes a Data Storage interrupt, the register state is altered as defined in *Table 9-6*, and instruction fetch and execution resume at the effective address specified in *Table 9-3 Interrupt Vector and Exception Conditions* on page 244.

Table 9-6. Registers Altered by a Data Storage Interrupt

Register	Bits	Setting Description
SRR0	0:63	Set to the effective address of the instruction that caused the exception condition.
SRR1	0:63	Set as defined in the PowerPC Architecture.
MSR	0:63	See *Table 9-40 Machine State Register (MSR) Bit-Settings Due to Interrupts* on page 290.
DSISR	0:31	Set as defined in the PowerPC Architecture.
DAR	0:63	Set as defined in the PowerPC Architecture.

9.5.3.1 *Data Address Breakpoint Exception*

The PPE supports the optional data address breakpoint facility as defined in *PowerPC Operating Environment Architecture, Book III*. This facility is controlled by the Data Address Breakpoint Register (DABR) and the Data Address Breakpoint Register Extension (DABRX). A data address breakpoint match occurs if any byte accessed matches the doubleword address specified in the upper 61 bits of the DABR. The lower three bits of the DABR and the DABRX control the conditions under which a Data Address Breakpoint Exception occurs. Each thread has an independent DABR. See *PowerPC Operating Environment Architecture, Book III* for more information about the Data Address Breakpoint facility.

For nonmicrocoded instructions, the PPE does not alter or access any bytes of storage when a DABR match occurs.

In this processor implementation, a DABR match occurs for an **stwcx.** or **stdcx.** instruction regardless of the reservation.

9.5.4 Data Segment Interrupt (x‘00..00000380’)

A Data Segment interrupt occurs when no higher-priority interrupts exists and the effective address of the data access cannot be translated to a virtual address. Data Segment interrupts are precise and are not maskable. The Data Segment interrupt is implemented as defined in the PowerPC Architecture with one exception. The Data Segment interrupt modifies the DSISR.

The PowerPC Architecture describes the processor behavior for the following situation as implementation-specific: an **stwcx.** or **stdcx.** instruction is executed and a Data Segment exception is generated. For this case, storage is not updated and the processor will take the Data Segment interrupt.

The PPE treats the Data Segment interrupt conditions as a context-synchronizing operation as defined in the PowerPC Architecture. When the PPE takes a Data Segment interrupt, the register state is altered as defined in *Table 9-7*, and instruction fetch and execution resume at the effective address specified in *Table 9-3 Interrupt Vector and Exception Conditions* on page 244.

Table 9-7. Registers Altered by a Data Segment Interrupt

Register	Bits	Setting Description
SRR0	0:63	Set to the effective address of the instruction that caused the exception condition.
SRR1	0:63	Set as defined in the PowerPC Architecture.
MSR	0:63	See *Table 9-40 Machine State Register (MSR) Bit-Settings Due to Interrupts* on page 290.
DSISR	0:5	Set to ‘0’.
	6	Set to ‘1’ on any store or **dcbz** instruction.
	7:9	Set to ‘0’.
	10	Set to ‘1’.
	11:31	Set to ‘0’.
DAR	0:63	Set as defined in the PowerPC Architecture.

9.5.5 Instruction Storage Interrupt (x‘00..00000400’)

An Instruction Storage interrupt (ISI) occurs when no higher-priority exception exists and the next instruction cannot be fetched. ISIs are precise and are not maskable. The ISI is implemented as it is defined by the PowerPC Architecture. In general, an Instruction Storage exception is caused by the following conditions:

- When the virtual address of the next instruction cannot be translated to a real address
- When an instruction fetch violates the storage protection as defined in the *PowerPC Operating Environment Architecture, Book III*.

  The storage protection is based on the [Ks] and [Kp] bits in the SLB, and the [PP] bits in the PTE when MSR[IR] is set to ‘1’. When MSR[IR] is set to ‘0’, an instruction storage protection violation occurs if either:

  – LPCR[LPES] bit 1 = ‘0’ and MSR[HV] = ‘0’

  – LPCR[LPES] bit 1 = ‘1’, MSR[HV] = ‘0’, and the instruction effective address is greater than or equal to the LPCR[RMLS]
- When an instruction fetch is attempted from a no-execute page (PTE[N] = ‘1’)
- When an instruction fetch is attempted with translation on (MSR[IR] = ‘1’) from a guarded page (PTE[G] = ‘1’)
- When an instruction is fetched from a no-execute segment (SLBE[N] = ‘1’)

The PPE treats the Instruction Storage interrupt conditions as a context-synchronizing operation as defined in the PowerPC Architecture. When the PPE takes an Instruction Storage interrupt, the register state is altered as defined in *Table 9-8*, and instruction fetch and execution resume at the effective address specified in *Table 9-3 Interrupt Vector and Exception Conditions* on page 244.

Table 9-8. Registers Altered by an Instruction Storage Interrupt

Register	Bits	Setting Description
SRR0	0:63	Set to the effective address of the instruction that the processor would have attempted to execute next if no exception condition were present. If the interrupt occurs due to a branch target fetch, SRR0 is set to the branch target.
SRR1	0:63	Set as defined in the PowerPC Architecture.
MSR	0:63	See *Table 9-40 Machine State Register (MSR) Bit-Settings Due to Interrupts* on page 290.
DSISR	0:31	Set as defined in the PowerPC Architecture.

9.5.6 Instruction Segment Interrupt (x‘00..00000480’)

An Instruction Segment interrupt (ISEG) occurs when no higher-priority exception exists and the fetch the next instruction cannot be performed. Instruction Segment interrupts are precise and are not maskable. The ISEG is implemented as defined by the PowerPC Architecture. In general, an ISEG is caused when the effective address of the next instruction cannot be translated to a virtual address.

The PPE treats the ISEG conditions as a context-synchronizing operation as defined in the PowerPC Architecture. When the PPE takes an ISEG, the register state is altered as defined in *Table 9-8*, and instruction fetch and execution resume at the effective address specified in *Table 9-3 Interrupt Vector and Exception Conditions* on page 244.

Table 9-9. Registers Altered by an Instruction Segment Interrupt

Register	Bits	Setting Description
SRR0	0:63	Set to the effective address of the instruction that the processor would have attempted to execute next if no exception condition were present. If the interrupt occurs due to a branch target fetch, SRR0 is set to the branch target.
SRR1	0:63	Set as defined in the PowerPC Architecture.
MSR	0:63	See *Table 9-40 Machine State Register (MSR) Bit-Settings Due to Interrupts* on page 290.

9.5.7 External Interrupt (x‘00..00000500’)

An External Interrupt occurs when higher-priority interrupts do not exist, an External Interrupt signal is asserted for one of the two execution threads, and External Interrupts are enabled. External Interrupts are system-caused and can be masked.

The PPE supports two kinds of External Interrupts—Direct and Mediated. Details about these interrupts are given in the following sections:

- *Section 9.6 Direct External Interrupts* on page 262.
- *Section 9.7 Mediated External Interrupts* on page 273.
- *Section 9.8 SPU and MFC Interrupts Routed to the PPE* on page 276.

9.5.8 Alignment Interrupt (x‘00..00000600’)

An Alignment interrupt occurs when no higher-priority interrupts exists and a data access cannot be performed for one of the following reasons:

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

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

- Cacheable (I = ‘0’) load/store instruction crossing a 32-byte boundary
- Cacheable (I = ‘0’) floating-point doubleword that is load/store word aligned, but not doubleword aligned
- Cacheable (I = ‘0’) load/store instruction that crosses a virtual page boundary

Additionally, if the DABR match feature is enabled, any non-VMU cacheable (I = ‘0’) load or store instruction crossing an 8-byte boundary causes an Alignment interrupt.

The PPE treats the Alignment interrupt conditions as a context-synchronizing operation as defined in the PowerPC Architecture. When the PPE takes an Alignment interrupt, the register state is altered as defined in *Table 9-10*, and instruction fetch and execution resume at the effective address specified in *Table 9-3 Interrupt Vector and Exception Conditions* on page 244.

Table 9-10. Registers Altered by an Alignment Interrupt

Register	Bits	Setting Description
SRR0	0:63	Set to the effective address of the instruction that caused the exception condition.
SRR1	0:63	SRR1 is set as defined in the PowerPC Architecture.
MSR	0:63	See *Table 9-40 Machine State Register (MSR) Bit-Settings Due to Interrupts* on page 290
DAR	0:63	Set as defined in the PowerPC Architecture.
DSISR	0:31	Set as defined in the PowerPC Architecture.

9.5.9 Program Interrupt (x‘00..00000700’)

A Program interrupt occurs when no higher-priority interrupts exist and one of the conditions listed below exist (as defined in the *PowerPC Architecture*):

- Floating-point enabled exception
- Illegal instruction
- Privileged instruction
- Trap

The PPE treats all floating-point exception modes, if enabled, as precise.

The PPE invokes the Illegal Instruction type of Program interrupt when it detects any instruction from the illegal instruction class, defined in *PowerPC User Instruction Set Architecture, Book I*. An **mtspr** or **mfspr** instruction with a reserved Special Purpose Register (SPR) selected causes an Illegal Instruction type of Program interrupt. In addition to these conditions, the PPE can be configured using the HID registers to cause an Illegal Instruction type of Program interrupt for the following condition:

- If HID0[en_attn] = ‘0’, the **attn** instruction causes an Illegal Instruction type of Program interrupt

The following cases list other implementation-specific causes of the Illegal Instruction type of Program interrupt (in addition to the cases listed in the architecture):

- If instructions are stored into the instruction stream (for example, self-modifying code), it is possible for an application error to cause a nonmicrocoded instruction to be sent to the microcode engine.
- Load or Store with Update invalid forms (when RA = ‘0’ or RA = RT)
- Load Multiple invalid forms (when RA is in the range of registers to be loaded or when RA = ‘0’)
- Load String invalid forms (when RA or RB is in the range of registers to be loaded, including the case in which RA = ‘0’, or when RT = RA or RT = RB).

  Note: *For the invalid forms where RA is in the range of registers to be loaded, the PPE completes the load up unto the point at which the range collision occurs. The interrupt is then taken at the point of collision.*

- Load or Store Floating-Point with Update invalid forms (when RA = ‘0’)
- **bcctr[l]** invalid form (BO[2] = ‘0’)

The PPE invokes the Privileged Instruction type of Program interrupt for the following cases (in addition to any cases listed in the architecture):

- A read or write is performed in problem state to a supervisor or hypervisor special purpose register.
- A read or write is performed in problem or supervisor state to a hypervisor special purpose register.
- A privileged instruction such as **rfid** or **hrfid** is executed without proper privileges.

The PPE treats the Program interrupt conditions as a context-synchronizing operation as defined in the PowerPC Architecture. When the PPE takes a Program interrupt, the register state is altered as defined in *Table 9-11*, and instruction fetch and execution resume at the effective address specified in *Table 9-3 Interrupt Vector and Exception Conditions* on page 244.

Table 9-11. Registers Altered by a Program Interrupt

Register	Bits	Setting Description
SRR0	0:63	SRR0 is set as defined in the PowerPC Architecture.
SRR1	0:63	SRR1 is set as defined in the PowerPC Architecture.
MSR	0:63	See *Table 9-40 Machine State Register (MSR) Bit-Settings Due to Interrupts* on page 290.

9.5.10 Floating-Point Unavailable Interrupt (x‘00..00000800’)

The Floating-Point Unavailable interrupt is implemented as defined in the PowerPC Architecture.

The PPE treats the floating-point interrupt as a context-synchronizing operation as defined in the PowerPC Architecture. When the PPE takes a Floating-Point Unavailable interrupt, the register state is altered as defined in *Table 9-12*, and instruction fetch and execution resume at the effective address specified in *Table 9-3 Interrupt Vector and Exception Conditions* on page 244.

Table 9-12. Registers by a Floating-Point Unavailable Interrupt

Register	Bits	Setting Description
SRR0	0:63	Set to the effective address of the instruction that caused the exception condition.
SRR1	0:63	SRR1 is set as defined in the PowerPC Architecture.
MSR	0:63	See *Table 9-40 Machine State Register (MSR) Bit-Settings Due to Interrupts* on page 290.

9.5.11 Decrementer Interrupt (x‘00..00000900’)

The Decrementer interrupt is implemented as defined by the PowerPC Architecture. Decrementer interrupts are system-caused. A Decrementer exception condition exists if the most-significant bit of the decrementer is set to ‘1’. A Decrementer interrupt is taken if the Decrementer interrupt is enabled (`MSR[EE]` = ‘1’).

The PPE treats the Decrementer interrupt as a context-synchronizing operation as defined in the PowerPC Architecture. When the PPE takes a Decrementer interrupt, the register state is altered as defined in *Table 9-13*, and instruction fetch and execution resume at the effective address specified in *Table 9-3 Interrupt Vector and Exception Conditions* on page 244.

Table 9-13. Registers Altered by a Decrementer Interrupt

Register	Bits	Setting Description
SRR0	0:63	Set to the effective address of the instruction that the processor would have attempted to execute next if no exception conditions were present.
SRR1	0:63	Set as defined in the PowerPC Architecture.
MSR	0:63	See *Table 9-40 Machine State Register (MSR) Bit-Settings Due to Interrupts* on page 290.

9.5.12 Hypervisor Decrementer Interrupt (x‘00..00000980’)

The PPE implements a hypervisor decrementer that is shared for both threads and a corresponding Hypervisor Decrementer interrupt. These facilities are part of the basic `LPAR` extension to the PowerPC Architecture. A Hypervisor Decrementer interrupt occurs when no higher-priority

exception exists, a Hypervisor Decrementer exception condition exists, and Hypervisor Decrementer interrupts are enabled. Hypervisor interrupts are enabled when the following expression is equal to ‘1’:

```
(MSR[EE] | NOT(MSR[HV])) & LPCR[HDICE]
```

A Hypervisor Decrementer exception condition exists if the most-significant bit of the hypervisor decrementer is set to ‘1’. Hypervisor Decrementer interrupts are system-caused and are maskable. If a thread is suspended, it will not be resumed by a Hypervisor Decrementer exception. If both threads are active and have interrupts enabled, then both threads will take the Hypervisor Decrementer interrupt.

The PPE treats the Hypervisor Decrementer interrupt as a context-synchronizing operation as defined in the PowerPC Architecture. When the PPE takes a Hypervisor Decrementer interrupt, the register state is altered as defined in *Table 9-14*, and instruction fetch and execution resume at the effective address specified in *Table 9-3 Interrupt Vector and Exception Conditions* on page 244.

Table 9-14. Registers Altered by a Hypervisor Decrementer Interrupt

Register	Bits	Setting Description
HSRR0	0:63	Set to the effective address of the instruction that the processor would have attempted to execute next if no exception conditions were present.
HSRR1	0:63	HSRR1[33] is set to ‘1’. **Note:** Setting HSRR1[33] to ‘1’ for this interrupt is a deviation from the *PowerPC Architecture* for this implementation. HSRR1[34:36] and [42:47] are set to ‘0’. All other bits are set to the corresponding value in the MSR.
MSR	0:63	See *Table 9-40 Machine State Register (MSR) Bit-Settings Due to Interrupts* on page 290.

9.5.13 System Call Interrupt (x‘00..00000C00’)

The System Call interrupt is implemented as defined in the PowerPC Architecture. System Call interrupts are caused by the **sc** instruction; they are precise and cannot be masked.

The PPE treats the execution of the **sc** instruction, which is the condition for the interrupt, as a context-synchronizing operation as defined in the PowerPC Architecture. When the PPE takes a System Call interrupt, the register state is altered as defined in *Table 9-15*, and instruction fetch and execution resume at the effective address specified in *Table 9-3 Interrupt Vector and Exception Conditions* on page 244.

The **sc** instruction has a LEV bit which, when set to ‘1’, causes the Hypervisor bit, `MSR[HV]`, to be set to ‘1’ so that the CBE processor runs in hypervisor mode.

Table 9-15. Registers Altered by a System Call Interrupt

Register	Bits	Setting Description
SRR0	0:63	Set to the effective address of the instruction that the processor would have attempted to execute next if no exception conditions were present.
SRR1	0:63	Set as defined in the PowerPC Architecture.
MSR	0:63	See *Table 9-40 Machine State Register (MSR) Bit-Settings Due to Interrupts* on page 290.

9.5.14 Trace Interrupt (x‘00..00000D00’)

A Trace interrupt occurs when no higher-priority interrupts exist, and any instruction except **rfid**, **hrfid**, **sc**, or **attn** completes successfully, and `MSR[SE]` is set to ‘1’; or a branch instruction completes successfully and `MSR[BE]` is set to ‘1’. The Trace interrupt is implemented as defined in the PowerPC Architecture. Trace interrupts are instruction-caused, precise, and cannot be masked. The PPE does not implement the example extensions (optional) to the trace facility as outlined in the Example Trace Extensions (Optional) section of *PowerPC Operating Environment Architecture, Book III*.

The PPE generates a Trace interrupt when an **mtmsrd** instruction sets `MSR[SE]` to ‘1’. In this case, `SRR0` is set to the effective address of the **mtmsrd** instruction instead of the next instruction that the processor would have attempted to execute if no exception condition were present. This occurs only if the prior state of `MSR[SE]` was set to ‘0’. If an **mtmsrd** instruction changes the `MSR[SE]` bit from ‘1’ to ‘0’, the PPE does not generate a Trace interrupt on the **mtmsrd** instruction.

The architecture specifies that the value of the `MSR[SE]` bit before an **mtmsrd** is executed is used to determine if a Trace interrupt is taken after the **mtmsrd** instruction is executed. This implementation does the opposite; the value after the **mtmsrd** instruction executes is used to determine if a Trace interrupt is taken after the **mtmsrd** instruction.

If Trace interrupts are enabled and an **mtctrl** instruction that disables a thread is issued, then the the thread will be disabled and the value of `SRR0` is undefined.

The PPE treats the Trace interrupt conditions as a context-synchronizing operation as defined in the PowerPC Architecture. When the PPE takes a Trace interrupt, the register state is altered as defined in *Table 9-16*, and instruction fetch and execution resume at the effective address specified in *Table 9-3 Interrupt Vector and Exception Conditions* on page 244.

Table 9-16. Registers Altered by a Trace Interrupt

Register	Bits	Setting Description
SRR0	0:63	Set to the effective address of the instruction that the processor would have attempted to execute next if no exception conditions were present. In the case of an **mtmsrd** instruction that causes a Trace interrupt by changing the state of MSR[SE] from ‘0’ to ‘1’, SRR0 is set to the effective address of the **mtmsrd** instruction.
SRR1	0:63	SRR1 bits [33:36], [42], and [44:47] are set to ‘0’. SRR1 bit [43] is set to ‘1’. All other bits are set to the corresponding value in the MSR.
MSR	0:63	See *Table 9-40 Machine State Register (MSR) Bit-Settings Due to Interrupts* on page 290.

9.5.15 VXU Unavailable Interrupt (x‘00..00000F20’)

A VXU Unavailable interrupt occurs when a higher-priority interrupt does not exist, an attempt is made to execute a VXU instruction, and MSR[VXU] is set to ‘0’. The VXU Unavailable interrupt is a PPE-specific interrupt. The interrupt is not defined in the *PowerPC Architecture*, but it is defined in the Vector/SIMD Multimedia Extension to the *PowerPC Architecture*. VXU Unavailable interrupts are instruction-caused, precise, and cannot be masked.

The PPE treats the VXU Unavailable interrupt as a context-synchronizing operation as defined in the *PowerPC Architecture*. When the PPE takes a VXU Unavailable interrupt, the register state is altered as defined in *Table 9-17*, and instruction fetch and execution resume at the effective address specified in *Table 9-3 Interrupt Vector and Exception Conditions* on page 244.

Table 9-17. Registers Altered by a VXU Unavailable Interrupt

Register	Bits	Setting Description
SRR0	0:63	Set to the effective address of the instruction that the processor would have attempted to execute next if no exception condition were present.
SRR1	0:63	SRR1[33:36] and [42:47] are set to '0'. All other bits are set to the corresponding value in the MSR.
MSR	0:63	See *Table 9-40 Machine State Register (MSR) Bit-Settings Due to Interrupts* on page 290.

For additional information about this interrupt, see the *PowerPC Microprocessor Family: Vector/SIMD Multimedia Extension Technolgy Programming Environments Manual*.

9.5.16 System Error Interrupt (x'00..00001200')

A System Error interrupt occurs when no higher-priority interrupts exists, a System Error interrupt signal is asserted for one of the execution threads, HID0[en_syserr] = '1', and `MSR[EE]` = '1' or `MSR[HV]` = '0'. The System Error interrupt is a PPE-specific interrupt. The interrupt is not defined in the PowerPC Architecture. System Error interrupts are system-caused and are enabled if the following expression is equal to '1':

```
(MSR[EE] | NOT(MSR[HV])) & HID0[en_syserr]
```

The PPE treats the System Error interrupt conditions as a context-synchronizing operation as defined in the PowerPC Architecture. When the PPE takes a System Error interrupt, the register state is altered as defined in *Table 9-18*, and instruction fetch and execution resume at the effective address specified in *Table 9-3 Interrupt Vector and Exception Conditions* on page 244.

Table 9-18. Registers Altered by a System Error

Register	Bits	Setting Description
HSRR0	0:63	Set to the effective address of the instruction that the processor would have attempted to execute next if no exception condition were present.
HSRR1	0:63	HSRR1 bit 33 is set to '1' and bits [34:36] and [42:47] are set to '0'. All other bits are set to the corresponding value in the MSR.
MSR	0:63	See *Table 9-40 Machine State Register (MSR) Bit-Settings Due to Interrupts* on page 290

The IOC Fault Isolation register (`IOC_FIR`) acts as a collection point for error conditions from several sources within the CBE processor but where these errors are external to the PPE. See the *Cell Broadband Engine Registers* document for more information about the `IOC_FIR`.

Bits in the `IOC_FIR` are set as a result of errors detected by the EIB, Bus Interface Controller (BIC), and IOC. A subset of the bits can cause a System Error interrupt in the PPE. The following IOC-detected errors can result in a System Error signal being asserted if the corresponding IOC Fault Isolation Error Mask register (`IOC_FIR_ErrMask`) bit is '0', the corresponding IOC Fault Isolation Checkstop Enable register (`IOC_FIR_ChkStop_en`) bit is '0', and the corresponding IOC Fault Isolation System Error Enable register (`IOC_SysErrEn`) bit is '1':

- The time that a CBE processor originated command to IOIF0 (if `IOC_FIR_ErrMask[yi5]` = ‘0’, `IOC_FIR_ChkStop_en[yi5]` = ‘0’, and `IOC_SysErrEn[Q0]` = ‘1’) or to IOIF1 (if `IOC_FIR_ErrMask[yi11]` = ‘0’, `IOC_FIR_ChkStop_en[yi11]` = ‘0’, and `IOC_SysErrEn[Q1]` = ‘1’) has been in the IOC outbound command queue exceeds the time specified in the MMIO register IOCmd Configuration (`IOC_IOCmd_Cfg`). See the *Cell Broadband Engine Registers* specification for more information about `IOC_IOCmd_Cfg`.
- The time that a command from IOIF0 (if `IOC_FIR_ErrMask[yi4]` = ‘0’, `IOC_FIR_ChkStop_en[yi4]` = ‘0’, and `IOC_SysErrEn[A0]` = ‘1’) or from IOIF1 (if `IOC_FIR_ErrMask[yi10]` = ‘0’, `IOC_FIR_ChkStop_en[yi10]` = ‘0’, and `IOC_SysErrEn[A1]` = ‘1’) has been in the IOC inbound command queue exceeds the time specified in `IOC_IOCmd_Cfg`.
- An ERR response was received on a command from the CBE processor to IOIF0 (if `IOC_FIR_ErrMask[yi3]` = ‘0’, `IOC_FIR_ChkStop_en[yi3]` = ‘0’, and `IOC_SysErrEn[E0]` = ‘1’) or from the CBE processor to IOIF1 (if `IOC_FIR_ErrMask[yi9]` = ‘0’, `IOC_FIR_ChkStop_en[yi9]` = ‘0’, and `IOC_SysErrEn[E1]` = ‘1’).

If a System Error occurs, both threads are signaled. If the System Error is signaled to a PPE thread, whether a system reset interrupt occurs first or if a System Error interrupt is taken depends on the settings of various SPR bits in the PPE. See *Section 9.5.1 System Reset Interrupt (Selectable or x‘00..00000100’)* on page 245 for more information about interrupts due to an external interrupt signal’s being asserted. If a System Error interrupt occurs, another System Error interrupt is not taken until the System Error signal is deasserted and is then asserted again.

The System Error signal goes through the power management logic in order to synchronize any change in this signal relative to power management state changes. The Pause(0) state is exited when a System Error interrupt occurs. See *Section 15 Power and Thermal Management* on page 423 for more information about power management.

9.5.17 Maintenance Interrupt (x‘00..00001600’)

The Maintenance interrupt is intended for hardware debugging purposes only. This interrupt is not intended for normal programming use. The Maintenance interrupt is a PPE-specific interrupt. The interrupt is not defined in the PowerPC Architecture. System-Caused Maintenance interrupts are enabled if the following expression is equal to ‘1’:

```
(MSR[EE] | NOT(MSR[HV])
```

Instruction-Caused Maintenance interrupts cannot be masked.

A System-Caused Maintenance interrupt cannot occur without an appropriate boundary-scan debug tool. A System-Caused Maintenance interrupt occurs when a higher-priority interrupt does not exist, a trace signal is asserted, and System-Caused Maintenance interrupts are enabled. The PPE has two trace signals. Each trace signal can be configured through the hardware-debug scan chain to trigger a System-Caused Maintenance interrupt on one or both threads.

An Instruction-Caused Maintenance interrupt occurs when a higher-priority interrupt does not exist, and the interrupt is triggered by one of the following:

1. A completing instruction matches one of two opcode compare and mask registers (opcode compare)
2. An internal instruction address breakpoint (IABR) that is accessed by the hardware-debug scan chain

3. An internal data address breakpoint (DABR) also accessed by the scan chain.

The internal hardware data address breakpoints (set using TDABR and TDABRX) are not the same as the software data address breakpoints (set using DABR and DABRX) defined in the PowerPC Architecture. The facilities are similar in both name and function; however, one is meant for software debug and the other is meant for hardware debug. The internal hardware instruction address breakpoint facility (set using TIABR and TIABRX) works identically to the hardware data address breakpoint except that it is triggered by instruction addresses rather than data addresses.

The PPE treats the Maintenance interrupt as a context synchronizing operation as defined in the PowerPC Architecture. When the PPE takes a Maintenance interrupt, the register state is altered as defined in *Table 9-19*, and instruction fetch and execution resume at the effective address specified in *Table 9-3 Interrupt Vector and Exception Conditions* on page 244. When a Maintenance interrupt occurs, HSRR1 reports all conditions that are true for the instruction taking the interrupt, not just the condition that caused it.

Table 9-19. Registers Altered by a Maintenance Interrupt

Register	Bits	Setting Description
HSRR0	0:63	Set to the effective address of the instruction that the processor would have attempted to execute next if no exception condition were present.
HSRR1	0:32	Set to the corresponding value in the MSR.
	33	Source of Maintenance interrupt 0 The interrupt is caused by an instruction (also called an instruction-caused interrupt). 1 The interrupt is caused by the system (also called a system-caused interrupt).
	34	Set to ‘0’.
	35:36	If the interrupt is an instruction-caused interrupt (HSRR1[33] = ‘0’): 10 Opcode 0 Compare 01 Opcode 1 Compare 11 Opcode 0 and Opcode 1 Compare **Note:** If a trace match is set up to cause a Maintenance interrupt, and also **iabr**, **dabr**, or opcode compare is set up, then when a Maintenance interrupt caused by an **iabr**, **dabr**, or opcode compare occurs, the opcode compare bits can be falsely set by a previous Maintenance interrupt caused by a trace match. If the interrupt is a system-caused interrupt (HSRR1[33] = ‘1’): 00 Reserved 01 Trace 0 input asserted 10 Trace 1 input asserted 11 Trace 0 and Trace 1 input asserted
	37	Set to ‘0’.
	38	Set to the corresponding value in the MSR.
	39:44	Set to ‘0’.
	45	Set to ‘1’ for a hardware data address breakpoint (HSRR1[33] is set to ‘0’) The breakpoint address is loadable only through scan. **Note:** The hardware data address breakpoint reported by this bit is not the same as the DABR defined in *Section 9.5.3.1 Data Address Breakpoint Exception* on page 249.
	46	Set to ‘1’ for a hardware instruction address breakpoint. (HSRR1[33] is set to ‘0’) The breakpoint address is loadable only through scan.
	47	Set to ‘0’.
	48:63	Set to the corresponding value in the MSR.
MSR	0:63	See *Table 9-40 Machine State Register (MSR) Bit-Settings Due to Interrupts* on page 290.

9.5.18 Thermal Management Interrupt (x‘00..00001800’)

A Thermal Management interrupt occurs when no higher-priority interrupts exist, a Thermal Management interrupt signal is asserted for one of the two execution threads, and the Thermal Management interrupts are enabled. Thermal Management interrupts are enabled if the following expression is equal to ‘1’:

```
((MSR[EE] | NOT(MSR[HV])) & (HID0[therm_intr_en]))
```

The Thermal Management interrupt is a PPE-specific interrupt. This interrupt is not defined in the PowerPC Architecture. Thermal Management interrupts are system-caused and can be masked.

The PPE treats the Thermal Management interrupt as a context-synchronizing operation as defined in the PowerPC Architecture. When the PPE takes a Thermal Management interrupt, the register state is altered as defined in *Table 9-20*, and instruction fetch and execution resume at the effective address specified in *Table 9-3 Interrupt Vector and Exception Conditions* on page 244.

Table 9-20. Registers Altered by a Thermal Management Interrupt

Register	Bits	Setting Description
HSRR0	0:63	Set to the effective address of the instruction that the processor would have attempted to execute next if no exception condition were present.
HSRR1	0:32	Set to the corresponding value in the MSR.
	33	Set to ‘1’.
	34:47	Set to ‘0’.
	48:63	Set to the corresponding values in the MSR.
MSR	0:63	See *Table 9-40 Machine State Register (MSR) Bit-Settings Due to Interrupts* on page 290.

Ten thermal sensors in the CBE processor are used to detect the temperature of various parts of the chip. The Thermal Management Control Unit (TMCU) provides software with an interface to these sensors. The TMCU can be configured to cause a thermal management interrupt to the PPE. There is a set of registers that provides status of the thermal sensors, a set of interrupt status registers indicating an interrupt condition, and a set of mask registers for the interrupts.

The Thermal Sensor Current Temperature Status Register One (`TS_CTSR1`) and Thermal Sensor Current Temperature Status Register Two (`TS_CTSR2`) provide the current temperature information using encoded values to represent specific temperature ranges for the ten thermal sensors. Larger values indicate higher temperatures. The Thermal Sensor Interrupt Temperature Register One (`TS_ITR1`) and Thermal Sensor Interrupt Temperature Register Two (`TS_ITR2`) contain the corresponding encoding for the temperature that causes a thermal management interrupt to the PPE. The Thermal Sensor Global Interrupt Temperature register (`TS_GITR`) contains a second interrupt temperature level. This register specifies one interrupt temperature level that applies to all sensors in the CBE processor. When the encoded temperature value in the `TS_CTSR1` or `TS_CTSR2` for a sensor is greater than or equal to the corresponding sensor’s interrupt temperature encoding in the `TS_ITR1` and `TS_ITR2`, the corresponding bit in `TS_ISR` is set. When the temperature encoding in the `TS_CTSR1` and `TS_CTSR2` register for any sensor is greater than or equal to the global interrupt temperature encoding in the `TS_GITR`, the corresponding status bit in `TS_ISR` is set. If any `TS_ISR` bit is '1' and the corresponding `TS_IMR` bit is '1', a thermal management interrupt signal is asserted to the PPE.

To clear the interrupt condition, privileged software should set the corresponding mask bit(s) in the `TS_IMR` to '0'. To enable a thermal management interrupt, privileged software should ensure the temperature is below the interrupt temperature for the corresponding sensor(s) and then perform the following steps.

1. Write a ‘1’ to the corresponding status bit in the `TS_ISR`.
2. Write a ‘1’ to the corresponding mask bit in the `TS_IMR`.

When the temperature is not below the interrupt temperature, enabling the interrupt can result in the generation of an immediate management interrupt.

See *Section 15 Power and Thermal Management* on page 423 for additional details on thermal management interrupts.

9.6 Direct External Interrupts

This section describes the Direct External Interrupts supported by the PPE.

As shown in *Table 9-1* on page 239, the sources or causes of Direct External Interrupts include:

- SPU Instructions—*Section 9.6.3* on page 268.
- MFC Commands—*Section 9.6.3* on page 268.
- Memory Interface Controller (MIC) Auxiliary Trace Buffer Full—*Section 9.6.4.2* on page 270.
- I/O Devices (IOIF0, IOIF1)—*Section 9.6.4.3* on page 270.
- I/O-Address Translation—*Section 9.6.4.4* on page 271.
- Element Interconnect Bus (EIB) Possible Livelock Detection—*Section 9.6.4.5* on page 272
- Token Manager—*Section 9.6.4.6* on page 273
- CBE Performance Monitor—*Section 9.6.4.7* on page 273.
- Software Interrupt, also called InterProcessor Interrupt (IPI)—*Section 9.6.2.4* on page 268.

The PPE receives these interrupts from:

- An IIC, by means of the EIB, from:
  - An SPU or MFC (all such packets are actually sent by the MFC).
  - An I/O device by means of the IOIF0 or IOIF1 interface.
  - The IIC, due to an event recorded by the MIC, EIB, Token Manager, I/O-Address Translation logic, or CBE Performance Monitor.
- Software interrupts from one PPE thread to another PPE thread. They are generated by an MMIO write to an Interrupt Generation Port (IGP) register, `IIC_IGP0` or `IIC_IGP1`, in the IIC.

As shown in *Figure 9-1* on page 240, the IICs receive and route these interrupts and interrupt status information to the PPE. The sections that follow describe the organization and handling of the Interrupts.

9.6.1 Interrupt Presentation

Interrupts generated by an SPU or MFC are sent to the IICs (or an external interrupt controller) as interrupt packets or dedicated signals on the EIB. External I/O devices send interrupts to the CBE processor using either interrupt packets or dedicated signals on an IOIF interface. When an IIC receives an interrupt packet it signals an External Interrupt to a PPE thread. If an external interrupt controller is used, the SPEs can be individually configured to send specific classes of interrupts to this external controller.

An IIC receives interrupt packets and signals an External Interrupt to the appropriate PPE thread. These IIC signals go through the power management logic in order to synchronize any changes in these signals relative to power-management state changes. The power management logic also detects that the PPE Pause(0) power management state should be exited when an External

Interrupt occurs. When an External Interrupt is signaled to a PPE thread, the type of interrupt taken depends on the settings of the special-purpose register (SPR) bits in the PPE. See *Section 15 Power and Thermal Management* on page 423 for more information about power management.

An incoming interrupt packet—generated within the CBE processor or external to it—destined for a PPE thread will have a Unit ID of x'E' or x'F', as shown in *Table 9-21*.

Table 9-21. Interrupt-Destination Unit ID

Destination Unit	Unit ID
PPE Thread 0	x'E'
PPE Thread 1	x'F'
External Interrupt Controller attached to IOIF1, if any	x'B'
External Interrupt Controller attached to IOIF0, if any	x'0'

The registers related to External Interrupt generation, routing, and presentation are summarized in *Section 9.6.2.1 IIC Register Summary* on page 263 and *Section 9.6.4 Other External Interrupts* on page 269. The Current Priority (`IIC_CPL0` and `IIC_CPL1`), Interrupt Pending Port (`IIC_IPP0` and `IIC_IPP1`), Interrupt Generation Port (`IIC_IGP0` and `IIC_IGP1`), IIC Interrupt Routing (`IIC_IRR`), and IIC Interrupt Status Register (`IIC_ISR`) are the most important. For details, see the *Cell Broadband Engine Registers* specification and the *Cell Broadband Engine Architecture* document.

9.6.2 IIC Interrupt Registers

9.6.2.1 *IIC Register Summary*

Table 9-22 summarizes the interrupt-related registers—one for each PPE thread—in the IICs, which themselves are part of the I/O Interface Controller (IOC). For details, see the *Cell Broadband Engine Registers* document and the *Cell Broadband Engine Architecture* document.

Table 9-22. IOC Interrupt Register Summary (Page 1 of 2)

Register Name	Mnemonic	Description	Read/ Write
Interrupt Registers *(Implementation-specific Registers)*			
IIC Thread 0 Interrupt Pending Port (non-destructive)	IIC_IPP0	Allows software to read the interrupt source and other information about pending interrupts. This register is sometimes referred to as the Interrupt Pending Port (IPP). After software reads this register destructively or non-destructively, the next highest interrupt is loaded into the register. When reading IIC_IPP0 non-destructively, the value of the associated IIC_CPL0 register is not updated with the interrupt in IIC_IPP0. When reading IIC_IPP0 destructively, the IIC_CPL0 register takes on the priority of the interrupt in IIC_IPP0. If a non-destructive read of the interrupt pending port that returns a valid interrupt is followed by another read of the interrupt pending port (destructive or non-destructive), the IIC returns the same data value unless another interrupt of a higher priority has been received by the IIC.	R
IIC Thread 0 Interrupt Pending Port (destructive)			R
IIC Thread 0 Interrupt Generation Port	IIC_IGP0	Allows privileged software to generate interrupts to PPE thread 0.	W
IIC Thread 0 Current Priority Level	IIC_CPL0	Allows software to mask interrupts of a specified priority for PPE thread 0. Software directly loads the CPL by means of MMIO write of the register or indirectly by means of destructive read of IIC_IPP0. When executing a destructive read of IIC_IPP0, the priority of the interrupt in IIC_IPP0 is written to the CPL if that interrupt is valid. Otherwise, the content is unchanged.	R/W
IIC Thread 1 Interrupt Pending Port (non-destructive)	IIC_IPP1	Same as IIC_IPP0, except for PPE thread 1.	R
IIC Thread 1 Interrupt Pending Port (destructive)			R
IIC Thread 1 Interrupt Generation Port	IIC_IGP1	Same as IIC_IGP0, except for PPE thread 1.	W
IIC Thread 1 Current Priority Level	IIC_CPL1	Same as IIC_CPL0, except for PPE thread 1.	R/W
IIC Interrupt Routing	IIC_IRR	Configures the priority and destination of certain interrupts reported in the IIC_ISR.	R/W
IIC_Interrupt Status	IIC_ISR	Records interrupt conditions from the Memory Interface, I/O interface, Element Interconnect Bus (EIB), Performance Monitoring, and Token Manager. If the IIC_ISR is non-zero, the IIC creates a class 1 interrupt to either the EIB, IIC_IPP0, or IIC_IPP1, depending on how the IIC_IRR is configured. After resetting the interrupt condition in the source unit, software can reset the interrupt by writing ‘1’ to the corresponding bit in the IIC_ISR. Writing this register must occur to confirm handling of interrupts in the IIC_ISR.	R/W

Table 9-22. IOC Interrupt Register Summary (Page 2 of 2)

Register Name	Mnemonic	Description	Read/ Write
Fault Isolation Registers *(Implementation-specific Registers)*			
IOC Fault Isolation	`IOC_FIR`	Records, sets, resets, and masks IOC faults. Enables checkstop interrupt for faults. `IOC_FIR_SysErrEnbl` is not part of other FIR registers; it is used to generate an enable for the System Error interrupt.	R/W
IOC Fault Isolation Register Set	`IOC_FIR_Set`		R/W
IOC Fault Isolation Register Reset	`IOC_FIR_Reset`		R/W
IOC Fault Isolation Error Mask	`IOC_FIR_ErrMask`		R/W
IOC Checkstop Enable	`IOC_FIR_ChkStpEnbl`		R/W
IOC System Error Enable	`IOC_FIR_SysErrEnbl`		R/W

9.6.2.2 *Current Priority Level*

There is one Current Priority Level (CPL) register for each PPE thread in the CBE processor. `IIC_CPL0` holds the current priority level for thread 0, and `IIC_CPL1` holds the current priority level for thread 1. The registers are intended to hold the priority level at which software is currently operating. The lower the numeric value of the priority field, the higher the priority level (highest priority corresponds to a numeric value of ‘0’). If the priority of an interrupt is numerically less than the priority in the CPL register, the interrupt is signaled to the PPE.

The priority-level value can be written explicitly by software to the CPL registers or indirectly by software doing a destructive read of a valid interrupt in the Interrupt Pending Port (IPP) register. Only the most-significant 5 bits of the CPL register priority field are implemented. Unimplemented bits of the interrupt pending port are read as ‘0’. This implies that `IIC_CPL0[Priority[0:3]]` is set to `IIC_IPP0[Priority[0:3]]`, and `IIC_CPL0[Priority[4]]` is set to ‘0’ when software performs a destructive read of a valid interrupt from `IIC_IPP0`. Likewise, `IIC_CPL1[Priority[0:3]]` is set to `IIC_IPP1[Priority[0:3]]`, and `IIC_CPL1[Priority[4]]` is set to ‘0’ when software performs a destructive read of a valid interrupt from `IIC_IPP1`.

Although the CPL registers have five bits, the IPP registers have only four bits. This extra bit in the CPL registers allows software to set the current priority level lower than any pending interrupt in the IPP registers. This allows for 16 priority levels to be used for these interrupts, whereas only 15 priority levels could be used if the CPL registers had only four bits. If a CPL register had four bits, the lowest priority (highest numeric value) in this register is all ones, and this masks an interrupt in the corresponding IPP register with the same priority value.

9.6.2.3 *Interrupt Pending Port*

The Interrupt Pending Port (IPP) registers allow software to read the interrupt packet data and other information about the highest priority interrupt pending for each PPE thread. The Thread 0 Interrupt Pending Port is `IIC_IPP0` and the Thread 1 Interrupt Pending Port is `IIC_IPP1`. *Table 9-23* shows the bit fields the registers (reserved fields are not shown).

Table 9-23. IIC_IPP0 and IIC_IPP1 Interrupt Pending Port Bit Fields

Bits	Name	Description
32	V	Interrupt Valid: 0 No interrupt pending. 1 Interrupt pending.
33	T	Interrupt Type: 0 SPE, external device, or external interrupt controller. 1 Thread 0 Interrupt Generation Port.
46:47	Class	Interrupt Class. Returns ‘0’s when T = ‘1’.
48:51	Src_Node_ID	Interrupt Source Node ID. Returns ‘0’s when T = ‘1’.
52:55	Src_Unit_ID	Interrupt Source Unit ID. Returns ‘0’s when T = ‘1’.
56:59	Priority	Interrupt Priority.

The interrupt source (ISRC) of an External Interrupt is designated as the two 4-bit fields, bits 48:55, shown in *Table 9-24*.

Table 9-24. Values for Interrupt Source

Interrupt Source	Most-Significant Four Bits of Interrupt Source (ISRC) (BIF Node ID)	Least-Significant Four Bits of Interrupt Source (ISRC) (BIF Unit ID)
SPU(0)	x‘4’	x‘4’
SPU(1)	x‘7’	x‘7’
SPU(2)	x‘3’	x‘3’
SPU(3)	x‘8’	x‘8’
SPU(4)	x‘2’	x‘2’
SPU(5)	x‘9’	x‘9’
SPU(6)	x‘1’	x‘1’
SPU(7)	x‘A’	x‘A’
IOC-IOIF1	x‘B’	x‘B’
IOC- IOIF0	x‘0’	x‘0’
IIC	not applicable	x‘E’

There are two types of read, one is non-destructive and the other is destructive. The destructive-read register is located at a doubleword offset to the non-destructive read address. Because of the IICs’ priority system, lower-priority interrupts will not be realized until the status is cleared (that is, reading the status by means of destructive reads).

The following description applies to interrupts for one PPE thread and that thread’s IPP register, CPL register, and IGP register.

The interrupt priority is 8 bits in the *Cell Broadband Engine Architecture*. The CBE processor implements the four most-significant bits of interrupt priority and ignores the four least-significant bits of interrupt priority in the IPP register. There are 16 interrupt priorities decoded from these

four bits. The IIC implements one interrupt-queue entry for each of the 16 priorities for each thread. In addition, for interrupts created by the MMIO writes to the IGP register, the IIC implements one queue entry for each of the 16 priorities.

When the priority of the highest-priority, valid interrupt in the interrupt-pending queue is higher (lower numeric value) than the priority in the current priority level, the External Interrupt signal to the PPE thread is activated. When the priority of the highest-priority, valid interrupt in the interrupt-pending queue is the same or lower (equal or higher numeric value) than the priority in the CPL register, the External Interrupt signal to the PPE thread is deactivated. If the PPE does not take an interrupt immediately because the External Interrupt is disabled or the interrupt is blocked because its priority is not higher than the priority in the CPL register, the interrupt is not lost. In this case, the interrupt is simply deferred until the External Interrupt is enabled and the priority in the CPL register is lowered.

After a destructive read of the IPP register, the IIC performs an interrupt-reissue transaction on the EIB if the IIC responded to an EIB interrupt command with a retry snoop response since the last interrupt-reissue transaction that the IIC performed. This means that the number of interrupt-reissue transactions might be less than the number of interrupt transactions that have been retried.

The IIC ignores priority in determining whether an interrupt-reissue transaction should be performed. Thus, the IIC can perform an interrupt-reissue transaction even when the priority of the interrupt that was given a retry response is different from the priority read from the interrupt pending port. In determining whether an interrupt-reissue transaction should be performed, the IIC also ignores the thread destination of the retried interrupt versus the thread whose interrupt pending port was read.

The IIC makes its determination of when an interrupt-reissue transaction occurred at a different point in an EIB transaction than when it determines its retry response for an interrupt command. The result is that more interrupt-reissue transactions might occur than necessary; however, there is not an interrupt transaction that received a retry response without a subsequent interrupt-reissue transaction on the EIB if there is no valid entry in the interrupt pending queue for the same priority as the retried interrupt.

The CBE processor does not support the *Cell Broadband Engine Architecture* optional interrupt-packet data field in the IPP.

The four most-significant bits of the interrupt source (ISRC) have the same value as the BIF Node ID of the interrupt source. The four least-significant bits have the same value as the BIF Unit ID of the interrupt source in most cases; however, for interrupts that the IIC generates by means of its Interrupt Routing register (IRR), the least-significant four bits of the ISRC are not related to a BIF unit ID, as shown in *Table 9-24* on page 266.

For interrupt packets from an IOIF, the IOC adjusts the source information when forwarding the interrupt packet to the EIB so that the IOC unit ID for that IOIF appears to be the source. Thus, an IPP register with an ISRC corresponding to IOC-IOIF0 in a specific BIF node represents an interrupt from a device on the IOIF0 connected to that IOC. An IPP value with an ISRC corresponding to IOC-IOIF1 in a specific BIF node represents an interrupt from a device on the IOIF1 connected to that IOC. When the MFC sends the interrupt packet on the internal EIB, the MFC inserts the ISRC that corresponds to itself.

9.6.2.4 *Interrupt Generation Port*

The Interrupt Generation Port (IGP) registers allows privileged software to generate an interrupt packet to a PPE thread. There is one IGP register for each PPE thread—Thread 0 Interrupt Generation Port (`IIC_IGP0`) register and Thread 1 Interrupt Generation Port (`IIC_IGP1`) register. Software can generate an interrupt packet to a PPE thread by storing to the PPE thread's IGP register. When the interrupt packet is read by means of the appropriate Interrupt Pending Port register (`IIC_IPP0` or `IIC_IPP1`), the interrupt packet data, class information, and ISRC are read as '0's. This interrupt packet does not need to be transmitted on the internal EIB since the IGP register and the destination of the interrupt packet are both in the same IIC. The least-significant 8 bits written to this register contains the interrupt priority; however, only the most-significant 4 bits of priority are implemented. For each priority level, a latch is used to represent a pending interrupt. A second store to the IGP with the same priority as a pending IGP interrupt results in no state change and, effectively, the interrupts are merged.

An IGP interrupt is treated like any other PPE External Interrupt, including the interrupt priority. Unlike other External Interrupts, however, this interrupt cannot be routed outside the CBE processor. For a multi-CBE-processor system, the registers can be memory mapped, and accessing the appropriate register will interrupt the desired PPE thread.

The `IIC_IGP0` and `IIC_IGP1` are write-only MMIO registers. The only field is the priority field, as shown in *Table 9-25*. Since this interrupt is meant to be a quick way to communicate with the other thread, it is recommended that the External Interrupt handler check this status before others.

Table 9-25. IIC_IGP0 and IIC_IGP1 Interrupt Generation Port Bit Field

Bits	Names	Descriptions
56:59	Priority	Interrupt Priority.

9.6.3 SPU and MFC Interrupts

Interrupt packets from the SPUs and MFCs are routed to the PPE by each MFC's Interrupt Routing Register (`INT_Route`). There are no implementation-dependent MFC interrupts; only the interrupts defined in the *Cell Broadband Engine Architecture* are supported. This section gives an overview of the status, masking, and routing of SPU and MFC interrupts. For details about these interrupts, see *Section 9.8 SPU and MFC Interrupts Routed to the PPE* on page 276.

The SPU and MFC interrupts routed to the PPE are independent from and unrelated to the event interrupts local to each SPE, which are described in *Section 18.3 SPU Interrupt Facility* on page 468.

9.6.3.1 *Status and Masking*

There are three interrupt status registers (`INT_Stat_class0`, `INT_Stat_class1`, and `INT_Stat_class2`) and mask registers (`INT_Mask_class0`, `INT_Mask_class1`, and `INT_Mask_class2`) in each MFC. There is an interrupt status register and an interrupt mask register for each class of interrupt (error, translation, and application).

9.6.3.2 *Routing*

The MFC Interrupt Routing (`INT_Route`) register, described in *Section 9.8.2.3 Interrupt Routing Register* on page 281, allows privileged software to select the PPE thread or external interrupt controller that is to service an interrupt. For each class of interrupt, the register contains a priority and interrupt destination.

Only the most-significant four bits of interrupt priority are implemented. The interrupt destination indicates which logical PPE thread or external interrupt controller is to receive interrupt packets for MFC interrupts of the corresponding interrupt class. The four most-significant bits of the interrupt destination have the same value as the BIF node ID of the interrupt destination. Refer to *Table 9-21* on page 263 for the least-significant four bit values of the desired destination. This table shows both the MFC setting of the destination IDs and the IIC interpretation of the destination IDs. When the MFC sends the interrupt packet on the internal EIB, the MFC inserts the interrupt source (ISRC) that corresponds to itself.

9.6.4 Other External Interrupts

The IIC Status register (`IIC_ISR`) acts as a collection point for interrupt conditions from the MIC, EIB, Token Manager, I/O Address Translation, and CBE Performance Monitor. There are internal signals from the MIC, EIB, Token Manager, I/O Address Translation logic, and CBE Performance Monitor, and these signals indicate the corresponding interrupt conditions to the IIC. When one of these interrupt signals is asserted, the corresponding bit in the `IIC_ISR` register is set to a ‘1’. When `IIC_ISR` becomes non-zero, the IIC creates a class 1 interrupt packet.

After resetting the interrupt condition in the MIC, EIB_Int, TKM_ISR, IOC_IO_ExcpStat, or the CBE Performance Monitor Status/Interrupt Mask register (pm_status), software can reset `IIC_ISR` by writing a ‘1’ to the corresponding bit position in `IIC_ISR`.

The MIC Auxiliary Trace Buffer Full interrupt condition can be reset by either:

- Setting `MIC_Ctl_Cnfg2[8]` = '0'
- If `MIC_CTL_CNFG2[7]` = '1', waiting for auxiliary trace current address to wrap so that it is no longer equal to the auxiliary trace max address
- If `MIC_CTL_CNFG2[7]` = '0', writing the auxiliary trace current address so that it is no longer equal to the Auxiliary trace max address.

Writing a ‘0’ to the corresponding bit has no effect on that bit.

9.6.4.1 *Routing*

The `IIC_IRR` register provides the priority and destination of the interrupt. The 4-bit priority field in the `IIC_IRR` register corresponds to the four bits of the priority field implemented in the `IIC_IPP0` and `IIC_IPP1` registers. If the interrupt is routed to the PPE thread in the same chip as the IIC, then the interrupt is placed in the appropriate interrupt pending queue for the thread indicated by the `IIC_IRR` register. If the interrupt is routed to some other destination, an interrupt is sent on the internal EIB. When the IIC sends the interrupt packet on the internal EIB, the IIC inserts the ISRC that corresponds to itself. The IIC’s unit ID in this ISRC is x‘E’, as shown in *Table 9-24* on page 266. If the `IIC_IRR` register destination node ID specifies some other node, then the inter-

rupt packet is sent to that node across the BIF protocol. The unit ID specifies a destination within a node or to an IOIF attached to the node, according to *Table 9-21* on page 263. If the destination is an external interrupt controller, the interrupt packet is sent across the appropriate IOIF.

The programmable fields in the `IIC_IRR` are shown in *Table 9-26*.

Table 9-26. IIC_IRR Routing Fields

Bits	Names	Descriptions
48:51	Priority	Interrupt Priority.
56:59	Dst_Node_ID	Destination node ID for interrupts in the IIC_ISR.
60:63	Dst_Unit_ID	Destination unit ID for interrupts in the IIC_ISR. Interrupt packets routed to IOIF0 have the same destination unit ID as IOC0. Those that are routed to IOIF1 have the same destination unit ID as IOC1. The valid values are: 0x'0' IOC 0 (IOIF0). 0x'B' IOC 1 (IOIF1). 0x'E' PPE thread 0. 0x'F' PPE thread 1.

9.6.4.2 *MIC Auxiliary Trace Buffer Full*

The CBE Performance Monitor (PM) facility provides a means of tracing various events and counts. The MIC provides a facility to store this trace in a region of memory. The memory trace region is considered full when the Auxiliary Trace Current Address Register is equal to the Auxiliary Trace Max Address. If the memory trace region is full and the MMIO register `MIC_Ctl_Cnfg2[8]` = '1', an MIC Auxiliary Trace Buffer Full signal to the IIC is asserted. When the MIC Auxiliary Trace Buffer Full signal is asserted, `IIC_ISR[60]` is set to '1'.

9.6.4.3 *I/O Devices*

I/O devices attached to the IOIF0 or IOIF1 interfaces can initiate and receive interrupts, and those interrupts can also be reissued. There are two types of interrupt transactions defined for the IOIF interfaces: Interrupt and Interrupt-reissue.

Interrupt transactions are initiated by sending an interrupt packet on the EIB. *Table 9-37* on page 286 shows the format of the I/O interrupt packet. After receiving such a packet, the IIC asserts the External Interrupt signal to the appropriate PPE thread. The first-level interrupt handler (FLIH) in that PPE thread then reads the Interrupt Pending Port Register (`IIC_IPP0` or `IIC_IPP1`) to determine the interrupting device.

If the target device cannot accept the interrupt, the interrupt command gets a retry response. The interrupting device must hold this interrupt pending until it sees an interrupt-reissue transaction. When an interrupt-reissue occurs, pending interrupts that received a retry since the last interrupt-reissue must be retransmitted if the BIF node ID field in the I/O identifier of the interrupt-reissue command sent by the CBE processor matches the destination BIF node ID of the pending interrupts.

For additional information about the I/O architecture in general, see *Section 7 I/O Architecture* on page 161.

9.6.4.4 *I/O Address Translation*

An I/O Address Translation Exception can occur due to either an I/O Segment Fault or I/O Page Fault. An I/O Segment Fault type of I/O Exception occurs for the following reason:

- There is an I/O translation that attempts to use an I/O Segment Table entry (IOSTE) that is not valid.

An I/O Page Fault type of I/O Exception occurs for the following reasons when an external device attempts a read or write on the IOIF with translation enabled:

- Assume a page size of 2^p is specified in IOSTE[PS]. The CBE processor supports I/O page-table (IOPT) entry format 1. See*Section 7.4 I/O-Address Translation* on page 176 for details of the IOST and IOPT. If the value of the most-significant bits (28 - p) of the 28 least-significant I/O address bits is greater than 512 times the IOSTE[NPPT], an I/O page-fault type of I/O exception occurs.
- The I/O address is translated using an I/O page-table entry whose IOID field does not match the IOID of the I/O device.
- The I/O address is translated using an I/O page-table entry whose page protection (PP) bits equal ‘00’, or whose PP bits equal ‘01’ and the IOIF transaction is a write, or whose PP bits equal ‘10’ and the IOIF transaction is a read.

When an I/O Exception occurs on an I/O access, the following information related to the fault is captured in the I/O exception status register (IOC_IO_ExcpStat):

- I/O Segment and I/O Page Number Address bits
- Access type (read or write)
- I/O Device ID

Once an I/O Exception occurs, IOC_IO_ExcpStat[V] is set to ‘1’. When IOC_IO_ExcpStat[V]=‘1’, errors that occur are not captured. After handling an I/O exception, software should set the valid bit to ‘0’ in order to enable capturing of error information about a subsequent I/O Exception.

If an I/O Segment Fault occurs and IOC_IO_ExcpMask[1]=‘1’, `IIC_ISR[61]` is set to ‘1’. If an I/O Page Fault occurs and IOC_IO_ExcpMask[2]=‘1’, `IIC_ISR[61]` is set to ‘1’. After handling an I/O Address Translation exception, software should write IOC_IO_ExcpStat[0] to '0' and write a ‘1’ to `IIC_ISR[61]` to reset this bit, It is recommended that software store to IOC_IO_ExcpStat to reset it first, store to `IIC_ISR` next, execute an **eieio** instruction, and then read IOC_IO_ExcpStat again to verify that another I/O Address Translation did not occur between the two stores. Assuming stores to these registers are Caching Inhibited and Guarded, the register bits are reset in program order.

IOC_IO_ExcpMask[1:2] is ANDed with associated fault conditions to set `IIC_ISR[61]`, but these mask bits do not gate the setting of the I/O Exception Status bits. As a result, if software were to set one mask bit to 1 and the other mask bit to 0, the MMIO register IOC_IO_ExcpStat could be set by the masked fault. In this case, as expected, software is not alerted to the occurrence. IOC_IO_ExcpStat captures and retains the exception status information for this masked fault. Typically, software uses the same value for both mask bits. When only one mask bit in IOC_IO_ExcpMask is set to '1', it is recommended that software frequently poll and reset IOC_IO_ExcpStat to avoid losing the exception status information for the enabled fault.

For additional details about I/O and I/O-address translation, see *Section 7 I/O Architecture* on page 161.

9.6.4.5 *EIB Possible Livelock Detection*

The EIB Possible Livelock Detection Interrupt detects a condition that may cause a livelock. It is a useful tool for programmers to use to detect whether they have created an unexpected bottleneck in their code during development. During normal running, this interrupt can hinder performance and should be disabled or masked. The EIB Possible Livelock Detection Interrupt signals that a possible livelock condition has been detected on the EIB due to over-utilization of the previous adjacent address match (PAAM) wait buffer. This can occur due to bad or malicious software or to frequent accesses by multiple units to common locations. Global commands that have a PAAM collision with an outstanding command are placed in the PAAM wait buffer. When this outstanding command gets its combined response, one command from the wait buffer that had a PAAM collision gets reflected on the EIB.

After the PAAM wait buffer completely fills, various things can happen, depending on how the MMIO register bits `EIB_AC0_CTL[13:15]`, the Livelock Avoidance Operating Mode field, are configured. When the wait buffer is full, the default mode ('110') is to allow the EIB command pipe to continue issuing EIB commands, but to force all commands that get a PAAM hit to receive a combined snoop response or retry.

When a wait buffer entry becomes available, a command that gets a PAAM hit is again placed in the wait buffer. There is logic that counts commands that exit the wait buffer and re-enter it due to yet another PAAM hit. This happens when multiple commands get a PAAM hit against the same cache line address within the same PAAM window. This is called a loopback. `EIB_AC0_CTL[4:5]`, the AC0 Wait Buffer Flush Delay field, defines an encoded threshold for these loopbacks. For example, when the default threshold of 64 loopbacks is hit, the command pipe is frozen until all the wait buffer commands are reflected and the wait buffer is empty. Then, normal operation resumes.

`EIB_AC0_CTL[10:11]` specifies threshold criteria for the wait buffer's becoming full again since it was last full. If the wait buffer completely fills again and the number of commands with the address modifier M bit = '1' that have occurred is less than the threshold, `EIB_Int[0]` is set to '1', provided that `EIB_AC0_CTL[13:16]` specifies that this interrupt is not disabled. `EIB_Int[0]` is not set to '1' if `EIB_AC0_CTL[13:15]` = '011', '100', '101', or '111'. If `EIB_Int[0]` changes from '0' to '1' and `EIB_AC0_CTL[16]` is set to '0', then `IIC_ISR[59]` is set to '1'.

When the EIB Possible Livelock Detection Interrupt occurs, privileged software should determine the cause of the interrupt and take appropriate action. Also, if `EIB_AC0_CTL[13:15]` is set to '010', privileged software must reset the MMIO register `EIB_Int[0]` for hardware to resume its use of the PAAM wait buffer.

Using the PAAM wait buffer will probably result in better performance for most applications. Overuse of the PAAM wait buffer is usually caused by poorly written or malicious software that creates a situation in which multiple devices simultaneously try to access the same 128-byte cache line.

9.6.4.6 *Token Manager*

The MIC and IOC provide feedback to the Token Manager on their command queues. The MIC and IOC have registers to hold programmable thresholds. The MIC and IOC provide the Token Manager information about the command queue levels relative to these programmable thresholds. The IOC provides this information for each IOIF. These queue levels are defined as follows, relative to the number of entries in the command queue: If feedback for a managed resource reaches Level 3, the corresponding TKM Interrupt Status bit is set. Whenever the MMIO register TKM Interrupt Status (TKM_ISR) has a non-zero bit and the corresponding enable bit in the MMIO register TKM_CR[59:63] enables the interrupt, an exception signal from the TKM to the IIC is asserted. When this signal is asserted, `IIC_ISR[TMI]` is set to ‘1’.

9.6.4.7 *CBE Performance Monitor*

The CBE Performance Monitor (PM) facility (an entirely different facility from the Performance Monitor facility described in the *PowerPC Architecture*) allows acquisition of performance information from multiple logic islands. It can be used to acquire a number of programmable counts. The PM facility provides four 32-bit event counters, which could alternatively be configured as eight 16-bit event counters. Up to 8 events from up to 8 different CBE logic islands can be counted simultaneously. A 32-bit interval timer is also provided.

A performance monitor exception can be caused by the trace buffer being full, the interval timer overflow, or an overflow of any of the eight event counters. The Performance Monitor Status/Interrupt Mask register (pm_status) is a dual function register. Writes to this address store data in the PM Interrupt Mask register. Reads from this address return PM Status register data and reset all PM Status register bits. Software can read the PM Status register to determine which exception occurred since the last read. The occurrence of a subsequent exception causes the corresponding bit of the PM Status register to be set to ‘1’. The PM Interrupt Mask register provides the ability to individually mask any one or more of these 10 conditions. If the logical AND of the PM Status register and the PM Interrupt Mask register is a non-zero value, and PM Control bit [0] is ‘1’, the PM interrupt signal to the IIC is asserted; otherwise, the PM interrupt signal to the IIC is deasserted. When the PM interrupt signal is asserted, `IIC_ISR[PMI]` is set to ‘1’.

9.7 Mediated External Interrupts

9.7.1 Mediated External Interrupt Architecture

The PPE implements a Mediated External Interrupt extension to the *PowerPC Architecture* for External Interrupts. On a shared processor (that is, a processor on which virtual partitions are dispatched), these External Interrupt enhancements can reduce interrupt latency. The new kind of External exception is called a *Mediated External Exception*. An External Interrupt that is caused by a Mediated External exception is called a *Mediated External Interrupt*. This section defines the extension; page 275 provides implementation details.

Bit 52 of the Logical Partition Control Register (LPCR) is defined as the “Mediated External Exception Request” [`MER`] bit. The value 1 means that a Mediated External exception is requested. The value ‘0’ means that a Mediated External exception is not requested.

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

On a shared processor with LPCR[LPES] bit 0 set to ‘0’, when redispatching a partition if an External Interrupt has occurred for the partition and has not yet been presented to the partition (by passing control to the operating system’s External Interrupt handler as described below), the hypervisor proceeds as follows:

- If the External Interrupt was Direct and the partition has MSR[EE] = ‘1’,
  - Set registers (MSR and SRR0/1, and the External Interrupt hardware registers as appropriate) to emulate the External Interrupt.
  - Return to the operating system’s External Interrupt handler.
  - Restore the partition's LPCR[MER].
- If the External Interrupt was Direct but the partition has MSR[EE] = ‘0’,
  - Return to the partition at the instruction at which it was interrupted.
  - Set LPCR[MER] to ‘1’.
- If the External Interrupt was Mediated and the partition now has MSR[EE] = ‘1’),
  - Set registers (MSR and SRR0/1, and External Interrupt hardware registers as appropriate) to emulate the original Direct External Interrupt.
  - Return to the operating system’s External Interrupt handler.
  - Set LPCR[MER] to ‘0’ if all External Interrupts (for the partition) now have been presented to the partition. Otherwise, restore the partition's LPCR[MER].

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

The MER bit has the same relationship to the existence of a Mediated External exception as bit zero of the Decrementer (DEC[0]) or Hypervisor Decrementer (HDEC[0]) has to the existence of a Decrementer or Hypervisor Decrementer exception—a value of ‘1’ indicates an exception. (See *PowerPC Architecture, Book III* for more information.) The exception effects of LPCR[MER] are considered consistent with the contents of LPCR[MER] if one of the following statements is true.

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

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

The following changes are made in the description of the External Interrupt.

- Direct External Interrupts are enabled if the value of this expression is ‘1’:

  `MSR[EE] | (^(LPCR[LPES] bit 0) & (^(MSR[HV]) | MSR[PR]))`

  In particular, if `LPCR[LPES]` bit 0 = ‘0’ (directing External Interrupts to the hypervisor), Direct External Interrupts are enabled if the processor is not in hypervisor state.

  Note: *Because the value of `MSR[EE]` is always ‘1’ when the processor is in problem state, the simpler expression: `MSR[EE] | ^(LPCR[LPES] bit 0 | MSR[HV])` is equivalent to the expression given above.*

  Mediated External Interrupts are enabled if the value of this expression is ‘1’:

  `MSR[EE] & (^(MSR[HV]) | MSR[PR])`

  In particular, Mediated External Interrupts are disabled if the processor is in hypervisor state.

- There is no relative priority between Direct and Mediated External exceptions. If an External Interrupt occurs when both kinds of External exceptions exist and are enabled, the exception that actually caused the interrupt can be either.

- When an External Interrupt occurs, the state is saved in `HSRR[0` and `1]` if `LPCR[LPES]` bit 0 = ‘0’, and in `SRR[0` or `1]` if `LPCR[LPES]` bit 0 = ‘1’. The state that is saved is independent of the contents of `LPCR[LPES]` bit 0 except as described in the next bullet (that is, the setting described for SRR0/1 in the current architecture continues to apply to SRR0/1 if `LPCR[LPES]` bit 0 = ‘1’, and applies instead to `HSRR[0` or `1]` if `LPCR[LPES]` bit 0 = ‘0’ except as modified as described in the next bullet).

  Similarly, the hypervisor's External Interrupt handler would use the HSPRGs as scratch registers if `LPCR[LPES]` bit 0 = ‘0’, and would use the SPRGs if `LPCR[LPES]` bit 0 = ‘1’.

- If `LPCR[LPES]` bit 0 = ‘0’, when an External Interrupt occurs `HSRR1[42]` is set to ‘1’ for a Mediated External Interrupt; otherwise it set to ‘0’. (If `LPCR[LPES]` bit 0 = ‘1’, when an External Interrupt occurs SRR1[42] is set to ‘0’ as in the current architecture.)

- The hypervisor must ensure that Mediated External Interrupts do not occur when `LPCR[LPES]` bit 0 = ‘1’. The hypervisor can accomplish this by setting `LPCR[MER]` to ‘0’ whenever it sets `LPCR[LPES]` bit 0 to ‘1’. (Mediated External Interrupts are disabled when the processor is in hypervisor state, and leaving hypervisor state is necessarily accomplished by means of a context synchronizing instruction, which, if `LPCR[MER]` = ‘0’, ensures that no Mediated External exception exists (see item 2).) If the hypervisor violates this requirement the results are undefined.

9.7.2 Mediated External Interrupt Implementation

The Mediated External Interrupt extension allows External Interrupts to be serviced by hypervisor software even when External Interrupts are disabled (`MSR[EE]` = ‘0’). This feature reduces the External Interrupt latency for applications running in a partitioned system.

This extension defines two conditions that cause an External Interrupt—the assertion of an External Interrupt signal or the hypervisor setting a Mediated External Interrupt request (by setting `LPCR[MER]` = ‘1’). An External Interrupt caused by the assertion of an External signal is

called a Direct External Interrupt. An External Interrupt caused by setting `LPCR[MER]` = ‘1’ is called a Mediated External Interrupt. Each of these conditions can be masked by the following equations:

- A Mediated External Interrupt is taken if the following expression is equal to ‘1’:

  `MSR[EE] & (NOT(MSR[HV]) | MSR[PR]) & HID0[extr_hsrr]`

- A Direct External Interrupt is taken if the following expression is equal to ‘1’:

  `MSR[EE] | ((NOT(LPCR[LPES] bit 0) & (NOT(MSR[HV]) | MSR[PR])) & HID0[extr_hsrr])`

For direct External Interrupts, the External Interrupt signals are required to remain asserted until the interrupt is taken. Deasserting an External Interrupt signal prior to the PPE taking the interrupt can lead to undefined results. After the PPE begins execution of the External Interrupt handler, the External Interrupt signal can be deasserted.

The PPE treats the External Interrupt conditions as a context-synchronizing operation as defined in the PowerPC Architecture. When the PPE takes an External Interrupt, the register state is altered as defined in *Table 9-7*, and instruction fetch and execution resume at the effective address specified in *Table 9-3 Interrupt Vector and Exception Conditions* on page 244.

Table 9-27. Registers Altered by an External Interrupt

Register	Bits	Setting Description
SRR0	0:63	Set to the effective address of the instruction that the processor would have attempted to execute next if exception conditions were not present. SRR0 is updated if LPCR[LPES] bit 0 = ‘1’ or HID0[extr_hsrr] = ‘0’.
HSRR0	0:63	Set to the effective address of the instruction that the processor would have attempted to execute next if exception conditions were not present. HSRR0 is updated if LPCR[LPES] bit 0 = ‘0’ and HID0[extr_hsrr] = ‘1’.
SRR1	0:63	If LPCR[RMI] = ‘1’ or HID0[extr_hsrr] = ‘0’, SRR1 is set as defined in the PowerPC Architecture. Otherwise, SRR1 is not altered.
HSRR1	0:63	If LPCR[RMI] = ‘0’ and HID0[extr_hsrr] = ‘1’, HSRR1[42] is set to ‘1’ if External Interrupt caused by LPCR[MER]. Otherwise, HSRR1[42] is set to ‘0’. HSRR1[33] is set to ‘1’. **Note:** Setting HSRR1[33] to ‘1’ for this interrupt is a deviation from the *PowerPC Architecture* for this implementation. All other HSRR1 bits are set as defined in the PowerPC Architecture for SRR1.
MSR	0:63	See *Table 9-40 Machine State Register (MSR) Bit-Settings Due to Interrupts* on page 290.

9.8 SPU and MFC Interrupts Routed to the PPE

This section describes the External Interrupts (*Section 9.6.3 SPU and MFC Interrupts* on page 268) caused by SPU instructions and MFC commands and routed by each SPE, to either the PPE or an external interrupt controller. Interrupts caused by SPU events are handled locally by each SPU and are described in *Section 18.3 SPU Interrupt Facility* on page 468.

9.8.1 Interrupt Types and Classes

Table 9-28 on page 277 summarizes the SPU and MFC interrupt types and classes. All SPU and MFC interrupts share the same PowerPC External Interrupt vector, defined in *Section 9.5.7 External Interrupt (x'00..00000500')* on page 251.

Table 9-28. SPU and MFC External Interrupt Definitions

Class[1]	Interrupt Type	Description	Enabling and Status Bit[2]
0	SPU Error (`SPU_Error`)	The SPU has encountered an error condition. These conditions include: • Invalid SPU instruction. • Illegal channel instruction. • Uncorrectable ECC error.	SE
	Invalid DMA Command (`Invalid_DMACmd`)	Software attempted to execute an MFC command with an invalid opcode, an MFC command not supported for the specific queue (for example, MFC atomic commands in the MFC proxy command queue), or an invalid form of an MFC command.	C
	DMA Alignment Error (`DMA_Alignment`)	Software attempted to execute a DMA command that is not aligned to a 128-byte boundary	A
	MFC Fault Isolation Register Interrupt (`MFC_FIR`)	MFC Fault Isolation Register (FIR) interrupt (see *Section 9.8.3.2* on page 284).	MF[3]
1	MFC Data Segment Error (`Data_Segment_Interrupt`)	A DMA effective address cannot be translated to a virtual address. (segment fault)	SF
	MFC Data Storage Error (`Data_Storage_Interrupt`)	A DMA effective address cannot be translated to a real address. (mapping fault)	MF[4]
	MFC Local Storage Address Compare Suspend on **get** (`Local-Storage Addr Comp Susp on get`)	A DMA LS address-compare stop from an LS write has occurred.	LG
	MFC Local Storage Address Compare Suspend on **put** (`Local-Storage Addr Comp Susp on put`)	A DMA LS address-compare stop from an LS read has occurred.	LP
2	Mailbox	An SPU writes to an SPU Write Outbound Interrupt Mailbox channel (`SPU_WrOutIntrMbox`).	M
	SPU Stop-and-Signal Instruction Trap (`SPU_Pgm_Stop`)	An SPU executes a stop-and-signal (**stop**) instruction.	S
	SPU Halt Instruction Trap (`SPU_Trapped`)	An SPU executes a halt conditional instruction, and the condition is met.	H
	Tag-Group Completion (`Tag_group_completion`)	A DMA command for a tag group has completed. The interrupt generation is dependent on the Proxy Tag-Group Query Mask Register (`Prxy_QueryMask`) and theProxy Tag-Group Query Type Register (`Prxy_QueryType`).	T
	SPU Inbound Mailbox Threshold (`SPU_mailbox_threshold`)	The number of valid entries in the SPU Inbound Mailbox queue has dropped from 1 to 0. See *Section 17* on page 441 for details.	B

1. The *Cell Broadband Engine Architecture* also names the classes as 0 = Error, 1 = Translation, 2 = Application. In the CBE processor, however, the MFC_FIR interrupt is moved from CBEA class 1 to CBE class 0. There are also differences in interrupt causes between the CBEA and CBE versions.
2. The bit field in the `INT_Mask_class` and `INT_Stat_class` registers (*Table 9-29* on page 278).
3. This is the MF bit in the `INT_Stat_class0` and `INT_Mask_class0` registers.
4. This is the MF bit in the `INT_Stat_class1` and `INT_Mask_class1` registers.

9.8.2 Interrupt Registers

Table 9-29 summarizes the interrupt-related registers in each SPE. The interrupt registers allow privileged software on the PPE to select which MFC and SPU exceptions are allowed to generate an external interrupt to the PPE. There are three interrupt mask and interrupt status registers, one for each class of interrupt (details of class membership are listed in *Table 9-28* on page 277). There are also fault-isolation registers and other registers that report details on specific types of interrupts.

For details about these registers, see the *Cell Broadband Engine Registers* and *Cell Broadband Engine Architecture* documents.

Table 9-29. SPE Interrupt Register Summary (Page 1 of 3)

Register Name	Mnemonic	Description	Read/ Write
Interrupt Registers (*Cell Broadband Engine Architecture Registers*)			
Class 0 Interrupt Mask Register	`INT_Mask_class0`	Enables the following interrupts: • MFC_FIR (MF) • SPU error (SE) • Invalid DMA command (C) • MFC DMA alignment (A)	R/W
Class 1 Interrupt Mask Register	`INT_Mask_class1`	Enables the following interrupts: • MFC local-storage address compare suspend on **put** (LP) • MFC local storage address compare suspend on **get** (LG) • MFC data-storage interrupt—mapping fault (MF) • MFC data segment interrupt—segment fault (SF)	R/W
Class 2 Interrupt Mask Register	`INT_Mask_class2`	Enables the following interrupts: • SPU Mailbox threshold (B) • Tag-group completion (T) • SPU Halt instruction trap (H) • SPU Stop-and-signal instruction trap (S) • Mailbox (M)	R/W
Class 0 Interrupt Status Register	`INT_Stat_class0`	Records status of the following interrupts: • MFC_FIR (MF) • SPU error (SE) • Invalid DMA command (C) • MFC DMA alignment (A)	R/W

Table 9-29. SPE Interrupt Register Summary (Page 2 of 3)

Register Name	Mnemonic	Description	Read/ Write
Class 1 Interrupt Status Register	`INT_Stat_class1`	Records status of the following interrupts: • MFC local-storage address compare suspend on **put** (LP) • MFC local storage address compare suspend on **get** (LG) • MFC data-storage interrupt—mapping fault (MF) • MFC data segment interrupt—segment fault (SF)	R/W
Class 2 Interrupt Status Register	`INT_Stat_class2`	Records status of the following interrupts: • SPU Mailbox threshold (B) • Tag-group completion (T) • SPU Halt instruction trap (H) • SPU Stop-and-signal instruction trap (S) • Mailbox (M)	R/W
Interrupt Routing Register	`INT_Route`	Configures the priority and destination of Class 0, 1, and 2 interrupts. Interrupts can be routed to either PPE thread or either IOIF interface.	R/W
Fault Isolation Registers *(Implementation-specific Registers)*			
MFC Fault Isolation	`MFC_FIR`	Records, sets, resets, and masks MFC faults. Enables checkstop interrupt for faults.	R/W
MFC Fault Isolation Set	`MFC_FIR_Set`		R/W
MFC Fault Isolation Reset	`MFC_FIR_Reset`		R/W
MFC Fault Isolation Error Mask	`MFC_FIR_Err`		R/W
MFC Fault Isolation Error Set Mask	`MFC_FIR_Err_Set`		R/W
MFC Fault Isolation Error Reset Mask	`MFC_FIR_Err_Reset`		R/W
MFC Checkstop Enable	`MFC_FIR_ChkStpEnbl`		R/W
Miscellaneous Registers *(Implementation-specific Registers)*			
MFC SBI Data Error Address	`MFC_SBI_Derr_Addr`	Records the destination address (as a slave device) when the SPE receives data that has an error and the error is recorded in `MFC_FIR` register	R
MFC Command Queue Error ID	`MFC_CMDQ_Err_ID`	Records the MFC command queue entry index number that caused a DMA bus transaction request error	R
MFC Status and Control Registers *(**Cell Broadband Engine Architecture Registers**)*			
MFC Address Compare Control	`MFC_ACCR`	Allows the detection of DMA access to a virtual page with the address compare (AC) bit set in the page-table entry (PTE) and a range within the LS.	R/W
MFC Data-Storage Interrupt Status	`MFC_DSISR`	Records status relating to data-storage interrupts (DSI) generated by the SMM.	R/W
MFC Data Address	`MFC_DAR`	Records the 64-bit effective address (EA) from a DMA command.	R/W

Table 9-29. SPE Interrupt Register Summary (Page 3 of 3)

Register Name	Mnemonic	Description	Read/ Write
MFC Command Data-Storage Interrupt Registers *(Implementation-specific Registers)*			
MFC Data-Storage Interrupt Pointer	`MFC_DSIPR`	Records the index (pointer) to the command that has an MFC data-storage interrupt (DSI) or an MFC data-segment interrupt. The cause of an MFC data-storage interrupt is identified in the `MFC_DSISR` register.	R
MFC Local Store Address Compare	`MFC_LSACR`	Records the LS address and LS-address mask to be used in the LS address compare operation selected by the `MFC_ACCR` register	R/W
MFC Local Store Compare Results	`MFC_LSCRR`	Records the LS address that triggered the compare, as well as the MFC command queue index of the DMA command that triggered the compare stop.	R
MFC Transfer Class ID	`MFC_TClassID`	Enables Transfer class ID and specifies issue quota and slot alternation.	R/W
MFC Command Error Register ***(Cell Broadband Engine Architecture Registers)***			
MFC Command Error	`MFC_CER`	Records the MFC command-queue-entry index of the command that generated the invalid DMA-command interrupt or the DMA-alignment interrupt.	R
SPU Error Mask Registers *(Implementation-specific Registers)*			
SPU Error Mask	`SPU_ERR_Mask`	Enables SPU invalid-instruction interrupt. (Illegal-operation detection and SPU halting are always enabled.)	R/W
MFC Control Register ***(Cell Broadband Engine Architecture Registers)***			
MFC Control	`MFC_CNTL`	Controls and reports decrementer status, and MFC command and command-queue status,	R/W

9.8.2.1 *Interrupt Mask Registers*

Table 9-29 lists three Interrupt Mask Registers (`INT_Mask_class0`, `INT_Mask_class1`, and `INT_Mask_class2`), one for each class of interrupts. Class 0 interrupts can be masked by means of the `INT_Mask_class0` register, as shown in *Table 9-30*.

Table 9-30. INT_Mask_class0 Bit Fields

Bits	Names	Descriptions
31	MF[1]	Enable for MFC_FIR interrupt.
61	SE	Enable for SPU error Interrupt.
62	C	Enable for invalid DMA command interrupt.
63	A	Enable for MFC DMA alignment interrupt.

1. This is the MF bit in the `INT_Stat_class0` and `INT_Mask_class0` registers, which is different than the MF bit in the `INT_Stat_class1` and `INT_Mask_class1` registers.

The `INT_Mask_class0[SE]` bit, shown in *Table 9-30*, can be individually masked by means of `SPU_ERR_Mask[I]` bit, as shown in *Table 9-31*.

Table 9-31. SPU_ERR_Mask Bit Fields

Bits	Name	Description
63	I	Invalid instruction interrupt enable: 0 Invalid instruction interrupt generation disabled. 1 Invalid instruction interrupt generation enabled. Generates a class 0 SPU error interrupt in INT_Stat_class0[61]. Illegal operation detection and SPU halting are always enabled.

Class 1 interrupts can be masked by means of the `INT_Mask_class1` register, as shown in *Table 9-32*.

Table 9-32. INT_Mask_class1 Bit Fields

Bits	Names	Descriptions
60	LP	Enable for MFC local-storage compare suspend on put interrupt.
61	LG	Enable for MFC local-storage compare suspend on get interrupt.
62	MF[1]	Enable for MFC data-storage interrupt (mapping fault).
63	SF	Enable for MFC data-segment interrupt (segment fault).

1. This is the MF bit in the `INT_Stat_class1` and `INT_Mask_class1` registers, which is different than the MF bit in the `INT_Stat_class0` and `INT_Mask_class0` registers.

Class 2 interrupts can be masked by means of the `INT_Mask_class2` register, as shown in *Table 9-33*.

Table 9-33. INT_Mask_class2 Bit Fields

Bits	Names	Descriptions
59	B	Enable for SPU mailbox threshold interrupt.
60	T	Enable for DMA tag group completion interrupt.
61	H	Enable for SPU halt instruction trap or single instruction step complete.
62	S	Enable for SPU stop-and-signal instruction trap.
63	M	Enable for mailbox interrupt.

9.8.2.2 *Interrupt Status Registers*

Each SPE has three Interrupt Status Registers (`INT_Stat_class0`, `INT_Stat_class1`, and `INT_Stat_class2`), which represent the status of the interrupts in the same bit position as the corresponding mask registers. A status-register bit that is set to ‘1’ represents an interrupt that is pending.

9.8.2.3 *Interrupt Routing Register*

SPE interrupts can be prioritized and routed using the `INT_Route` register, as shown in *Table 9-34*.

Table 9-34. INT_Route Bit Fields

Bits	Names	Descriptions
0:3	Class 0 priority	Priority for class 0 interrupts.
8:15	Class 0 destination	Destination ID for class 0 interrupts.
16:19	Class 1 priority	Priority for class 1 interrupts.
24:31	Class 1 destination	Destination ID for class 1 interrupts.
32:35	Class 2 priority	Priority for class 2 interrupts.
40:47	Class 2 destination	Destination ID for class 2 interrupts.

Each class priority is four bits (the highest priority is 0, and the lowest priority is 15). The destination consists of two values: the four most-significant bits are the BIF node ID of the interrupt destination, and the four least-significant bits specify the destination ID of the target unit, as shown in *Table 9-35*.

Table 9-35. CBE Unit Values

Value	Destination ID
0x'0'	External Interrupt controller attached to IOIF0, if any.
0x'B'	External Interrupt controller attached to IOIF1, if any.
0x'E'	PPE thread 0.
0x'F'	PPE thread 1.

9.8.3 Interrupt Definitions

The sections below gives details on Interrupts related to storage protection and MFC Fault-Isolation Register Interrupts. For details about other SPU and MFC Interrupts and the interrupt-generation process, see the *Cell Broadband Engine Architecture*.

9.8.3.1 *Storage-Protection Errors*

The SPE's Synergistic Memory Management Unit (SMM) checks for storage protection errors during both virtual- and real-mode address translation, although real-mode checking is very limited. The DMA controller (DMAC) also checks for LS-address compares. *Table 9-36* summarizes how the contents of the `INT_Stat_class1` and `MFC_FIR` registers are affected by the results of these checks.

Table 9-36. Exceptions Generated by Translation, MMIO, and Address Matches (Page 1 of 2)

Exception Name	INT_Stat_class1 Register			MFC_DSISR Register	DMAC Suspend	Data Error (DERR)	MFC_FIR Register
	LP or LG Bits[60:61]	MF (DSI) Bit[62]	SF Bit[63]				
SLB Segment Fault			X				
Atomic access to cache-inhibited page when translation is turned on		X		A bit[37]	X		

Table 9-36. Exceptions Generated by Translation, MMIO, and Address Matches (Page 2 of 2)

Exception Name	INT_Stat_class1 Register			MFC_DSISR Register	DMAC Suspend	Data Error (DERR)	MFC_FIR Register
	LP or LG Bits[60:61]	MF (DSI) Bit[62]	SF Bit[63]				
Atomic access to cache-inhibited page when translation is turned off (Real Mode)		X		A bit[37]	X		
Page protection violation (PP)		X		P bit[36]			
LS address compare (reported by DMAC and not reported as SMM exception)	X				X		
Data address compare (DAC)		X		C bit[41]	X		
TLB page fault (hardware or software)		X		M bit[33]			
Parity error detected during SLB/TLB translation operation					X		SLB or TLB bits [43:44}
Parity error detected during MMIO read to SLB or TLB array (only reported through DERR bit attached to the MMIO data returned to the requester)						X	

For all translation exceptions reported by the SMM, except for parity errors detected on Segment Lookaside Buffer (SLB) or Translation Lookaside Buffer (TLB) MMIO reads, the 64-bit EA is saved to the `MFC_DAR` register. If it is a DSI exception, information about which kind of DSI error and other related information is written into the `MFC_DSISR` register, and the MFC command-queue index is saved to the MFC Data Storage Interrupt Pointer Register (`MFC_DSIPR`). The `MFC_DAR`, `MFC_DSIPR`, and `MFC_DSISR` are locked to the value of the current outstanding exception or miss.

The SMM only supports one outstanding translation miss or exception, excluding MMIO read parity errors. The SMM has the ability to serve hits under one outstanding translation miss or exception. After the first outstanding miss or exception is detected by the SMM, any subsequent exception or miss of a DMA translation request is reported back to the DMAC as a miss. The DMAC sets the corresponding MMU dependency bit associated to the MFC command and stalls the command until the MMU dependency bit is cleared. When the outstanding miss or exception condition is cleared, all 24 (16-entry MFC SPU Command Queue and 8-entry MFC Proxy Command Queue) MMU dependency bits are cleared, and the DMAC re-sends the oldest MFC command to the SMM for translation. Exceptions are cleared by setting the Restart (R) bit in the `MFC_CNTL` register. Setting `MFC_CNTL[R]` also unlocks the `MFC_DAR` and `MFC_DSISR` registers and allows the SMM to report a new exception condition. The `MFC_CNTL[SC]` bit should be cleared by privileged software (after clearing the `MFC_CNTL[SM]` bit) to unsuspend the DMAC only after all exception conditions have been cleared. The `MFC_DSIPR` register is unlocked by reading the register or purging the command queue.

When a 16-KB DMA transfer crosses page boundary and causes a page fault, it is not restarted from the beginning; instead, the transfer is resumed where it left off. The DMA controller unrolls DMA transfers into cache-line-sized transfers. When a fault occurs, the DMA controller only knows which transfers are still outstanding. It does not retain the original transfer request, so is can only resume the transfer where it left off.

LS address compare exceptions occur when the LS is accessed within the address range specified in the `MFC_LSACR` register. The DMAC writes the `MFC_LSCRR` register with the LS address that met the compare conditions set in the `MFC_LSACR` register along with the MFC command queue Index and the DMA command type. This exception causes the DMAC to suspend. To clear the error, the `LP` or `LG` bit in the `INT_Stat_class1` MMIO register should be cleared and the DMAC unsuspended. See the MFC Local Store Compare Results Register (`MFC_LSCRR`) and the MFC Local Store Address Compare Register (`MFC_LSACR`) in the *Cell Broadband Engine Registers* document for more information.

SLB segment faults occur when there is no matching ESID in the array for the given EA from the translation request, or the valid bit is not set in the array for the matching `ESID` entry. When this fault occurs, the `MFC_DAR` register contains the 64-bit EA, as described above. The `MFC_DAR` register and the SLB array can be read by means of MMIO to determine which entry to replace in the SLB. Then, the entry should be invalidated with an MMIO `SLB_Invalidate_Entry` or `SLB_Invalidate_All` write, followed by `SLB_Index`, `SLB_ESID`, and `SLB_VSID` writes to put the proper entry into the SLB. The `INT_Stat_class1[SF]` should then be cleared and the `MFC_CNTL[R]` bit set to resume normal SMM and DMAC operation.

All DSI exceptions are defined in the *PowerPC Operating Environment Architecture, Book III*. When these exceptions occur, the SMM writes the 64-bit EA to the `MFC_DAR` register and records the DSI exception type in the `MFC_DSISR` register, as shown in *Table 9-36* on page 282. This table also shows which DSI exceptions cause the DMAC to suspend. `MFC_DSISR`[S] is also set if the DMAC translation request was a **put[rlfs]**, **putll[u]c**, or **sdcrz** operation. The SMM reports the DSI to the SBI unit to set the `INT_Stat_class1`[MF] bit. The `MFC_DAR` and `MFC_DSISR` registers and the SLB and TLB arrays may be read by means of MMIO to determine the type of DSI fault and the method for fixing it. After the error condition has been fixed, the `INT_Stat_class1[MF]` bit should be cleared and the `MFC_CNTL[R]` bit set.

SLB and TLB parity errors may be recoverable by replacing the faulty entry by means of MMIO. Parity errors detected during address translation set bits in the `MFC_FIR` register, as shown in *Table 9-36* on page 282. See the *Cell Broadband Engine Registers* document for more information on FIR registers. Only parity errors detected on an address translation cause the `MFC_DAR` to be locked with the EA value captured in it. After the error has been fixed, the `MFC_FIR` bits should be cleared and the `MFC_DSIPR` register written to restart the SMM and unsuspend the DMAC. Parity errors detected during MMIO reads on the SLB or TLB cause the `DERR` bit to be set on the EIB when the MMIO data is returned to the requester. The SMM can support an unlimited number of MMIO read parity errors because they do not lock the SMM or DMAC interrupt registers or affect address translation.

9.8.3.2 *MFC Fault-Isolation Register Interrupts*

The MFC Fault Isolation Register (`MFC_FIR`) Interrupt is controlled by the `MF` bit in the `INT_Stat_class0` and `INT_Mask_class0` registers. It is generated in the following cases:

- *DMA Command Alignment Error:*

  - Transfer size which is:
    - greater than 16 KB.
    - size neither partial nor quadword.
    - **sndsig** size not 4 bytes.
    - partial check for effective address failed.
  - List transfer size which is:
    - greater than 2 K list elements.
    - bits [13:15] non-zero.
  - LS address (LSA) in which:
    - bits [28:31] are not equal to the effective address.
    - bits [60:63] for **put**, **get**, or **sndsig** are not all ‘0’s for quadword transfers.
  - List address in which the lower three bits are non-zero.
- *DMA Command Error:*
  - Atomic command received while one is still pending in the queue.
  - List or atomic commands present in the MFC Proxy Command Queue.
  - Start modifier present in the MFC SPU Command Queue.
  - Upper 8 bits of opcode are not all ‘0’.
  - Invalid opcode.
  - Transfer tag (0:10) is non-zero.

9.8.4 Handling SPU and MFC Interrupts

When an SPE receives an interrupt request from its SPU, DMA Controller (DMAC), or Synergistic Memory Management Unit (SMM), it sends an interrupt packet to an IIC, as shown in *Figure 9-1* on page 240. The SPE sends an address-only transaction. There is no data packet associated with the interrupt packet. After receiving such a packet, the IIC asserts the External Interrupt signal to the appropriate PPE thread. The first-level interrupt handler (FLIH) in that PPE thread then reads the Interrupt Pending Port Register (`IIC_IPP0` or `IIC_IPP1`) to determine the interrupt class and source. Then, the PPE reads the SPU Interrupt Status Registers for each class of SPU interrupt (`INT_Stat_class0`, `INT_Stat_class1`, and `INT_Stat_class2`) to determine the cause of the interrupt.

9.8.4.1 *Interrupt Packets*

Table 9-37 shows the format for an address-only interrupt packet. Bits 24 through 31 of the interrupt packet’s address field identify the destination of the interrupt. Interrupts should not be processed by a device unless bits 24:31 of the interrupt packet’s address field match its device ID (that is, its BIF node ID concatenated with its unit ID). The address (bits 0:63) is not used for an interrupt-reissue command and is undefined in that case.

Only the target device can reject an interrupt packet. In that case, the BIF device that issued the interrupt packet must hold the interrupt pending until the target device issues an interrupt-reissue command. All BIF devices must re-send the pending interrupts if the BIF node field in the transfer

tag of the interrupt-reissue command matches the BIF node of the pending interrupts. Such pending interrupts must be resent after the combined snoop response of the interrupt-reissue command. Although the resulting performance may be less than optimal, BIF devices can optionally re-send the pending interrupts even if the BIF node field in the transfer tag of the interrupt-reissue command does not match the BIF node of the pending interrupts. Devices other than the target must provide a null snoop response for the interrupt transaction. All devices must provide a null or acknowledgment snoop response for the interrupt-reissue transaction. However, the master must give an acknowledgment snoop response.

Table 9-37. Format for Interrupt Packet

Bits	Width	Definition
Address (0:63)		
0:21	22	Reserved.
22:23	2	Interrupt class.
24:27	4	Destination BIF node ID.
28:31	4	Destination unit ID.
32:39	8	Interrupt priority.
40:63	24	Data.
Transfer Size (0:5)		
0	1	Reserved.
1:4	4	Source unit ID (typically, the same as bits 4 through 7 of the transfer tag).

As with all commands on the BIF, the device that issues an interrupt command uses its device ID as part of the transfer tag. The device is also required to put the unit ID of the device that requested the interrupt in the transfer size field of the interrupt command. In most cases, the unit ID in the size field matches the unit ID in the transfer tag. When the two values are different, the device that issues the interrupt command does so on behalf of the device identified by the unit ID in the transfer size field. This is referred to as an interrupt proxy. A device can only be an interrupt proxy for other devices within its BIF node.

The interrupt class is identified by bits 22:23 of the address. The meaning of the interrupt class is defined by the source of the interrupt and is typically provided to software as an indication of the type of interrupt. See *Book III* of the *Broadband Processor Architecture* for the use of the interrupt class.

The interrupt priority is identified by bits 32:39 of the address. The interrupt priority is typically used by an interrupt controller to determine if this interrupt should be presented to the processor, or to the device, or rejected. See *Book III* of the *Broadband Processor Architecture* for the use of the interrupt priority.

For additional information about the I/O architecture in general, see *Section 7 I/O Architecture* on page 161.

9.8.4.2 *Clearing Interrupts*

An interrupt status bit that has been set to ‘1’ in the Interrupt Status Registers (`INT_Stat_class0`, `INT_Stat_class1`, and `INT_Stat_class2`) must be cleared after the corresponding interrupt is handled. The only case in which this is not true is for class 1 interrupts in which the MFC Data-

Storage Interrupt Status Register (MFC_DSISR) needs to be cleared to ‘0’ before any of the status register bit for class 1 are cleared. Also, the MFC_CNTL[R] bit needs to be set to ‘1’ to resume DMA operation after status-register bits have been cleared.

Software should not clear the source of an interrupt until the Interrupt Status Register (INT_Stat_class0, INT_Stat_class1, and INT_Stat_class2) bits are read and cleared. If an Interrupt Status Register is cleared on the same cycle as the interrupt, the source sends a signal to the SBI, and this interrupt source can be lost. In the case of a MFC Fault Isolation Register (MFC_FIR) class 0 interrupt, software should clear the FIR bits first and then clear the Interrupt Status Register. If there is any FIR bit set when the Interrupt Status Register is cleared, the SBI sends a new interrupt packet.

9.9 Thread Targets for Interrupts

Table 9-38 on page 287 summarizes the PPE thread targets of the various interrupt types. The non-recoverable Checkstop error (which is not an interrupt) is described in *Section 9.15* on page 291.

Table 9-38. PPE-Thread Target for Interrupts

Exception Type	Interrupt	Target PPE Thread
Instruction-Caused (Synchronous)	All instruction-caused interrupts	Thread causing interrupt
System-Caused (Asynchronous)	System Reset, when a thread is started or restarted	Selectable by software
	System Reset, at Power-On Reset (POR)	Thread 0
	Machine Check[1]	Both threads
	System Error	One or both threads, depending on the error
	Decrementer	Thread causing interrupt
	Hypervisor Decrementer	Both threads
	Thermal Management	Selectable by software
	Maintenance (instruction-caused)	One or both threads, depending on the error
	External Interrupt (direct or mediated)[2]	Selectable by software

1. If HID0[en_prec_mchk] = ‘1’ (“precise”), only the thread that causes a Machine Check interrupt will take the interrupt, and the interrupt will appear to be instruction-caused (synchronous) from the viewpoint of that thread (see *Section 9.5.2* on page 246). If HID0[en_prec_mchk] = ‘0’ (“imprecise”), both threads will take a Machine Check interrupt, and the interrupt will appear to be system-caused (asynchronous) to both threads. A suspended thread will not resume.
2. See *Section 9.5.7* on page 251 for the difference between Direct and Mediated External Interrupts.

9.10 Interrupt Priorities

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

Table 9-39. Priority Order of Interrupt Conditions (Page 1 of 2)

I. System Reset Interrupt (highest priority exception)

II. Machine Check Interrupt

III. Instruction-Dependent

A. Fixed-Point Loads and Stores

1. Data Storage interrupt (DSI) caused by the DABR
2. Alignment interrupts caused by fixed-point load and store instructions (**lwarx**, **ldarx**, **stwcx.**, **stdcx.**, **lmw**, or **stmw**).
3. DSI caused by a logical partitioning (LPAR) error in real mode (MSR[HV] = ‘0’ and LPCR[HDICE] = ‘0’)
4. Data Segment interrupt
5. DSI caused by a page fault
6. Alignment interrupts (referencing caching-inhibited storage)
   a. Alignment interrupts caused by fixed-point load instructions that reference caching-inhibited storage.
   b. Alignment interrupts caused by fixed-point store instructions that reference caching-inhibited storage, if the C-bit in the page table entry is set to ‘1’.
7. DSI caused by a page-protection violation
8. Alignment interrupt caused by a fixed-point store instruction referencing caching-inhibited storage, if the C-bit in the page table entry is set to ‘0’.
9. DSI caused by the ACCR
10. Trace interrupt

B. Floating-Point Loads and Stores

1. Floating-Point Unavailable interrupt
2. DSI caused by the DABR
3. Alignment interrupt caused by a floating-point load and store instruction that is not word aligned.
4. DSI caused by an LPAR error in real mode (MSR[HV] = ‘0’ and LPCR[HDICE] = ‘0’)
5. Data Segment interrupt
6. DSI caused by a page fault
7. Alignment interrupts (referencing caching-inhibited storage)
   a. Alignment interrupt caused by a double-precision floating-point load instruction that is word-aligned, but not doubleword-aligned, referencing caching-inhibited storage.
   b. Alignment interrupt caused by a double-precision floating-point store instruction that is word-aligned, but not doubleword-aligned, referencing caching-inhibited storage and the C bit in the page table entry is set to ‘1’.
8. DSI caused by a page-protection violation
9. Alignment interrupt caused by a double-precision floating-point store instruction that is word-aligned, but not doubleword-aligned, referencing caching-inhibited storage, and the C bit in the page table entry is set to ‘0’.
10. DSI caused by the ACCR
11. Trace interrupt

C.Other Floating-Point Instructions

1. Floating-Point Unavailable interrupt
2, Precise-Mode Floating-Point Enabled exceptions type of Program interrupt
3. Trace interrupt

D. VXU Loads and Stores

1. VXU Unavailable interrupt
2. DSI caused by the DABR
3. DSI caused by an LPAR error in real mode (MSR[HV] = ‘0’ and LPCR[HDICE] = ‘0’)
4. Data Segment interrupt
5. DSI caused by a page fault
6. Alignment interrupts (referencing caching-inhibited storage)

Table 9-39. Priority Order of Interrupt Conditions (Page 2 of 2)

a. Alignment interrupts caused by VSU load and store (left or right) instructions that reference caching-inhibited storage.
b. Alignment interrupts caused by VSU load and store (left or right) instructions that reference caching-inhibited storage, if the C-bit in the page table entry is set to ‘1’.

7. DSI caused by a page-protection violation
8. Alignment interrupt caused by a VSU load and store (left or right) instruction referencing caching-inhibited storage, if the C-bit in the page table entry is set to ‘0’.
9. DSI caused by the ACCR
10. Trace interrupt

E. Other VXU Instructions
1. VXU Unavailable interrupt
2. Trace interrupt

F. **rfid**, **hrfid**, and **mtmsr**[**d**]
1. Precise-Mode Floating-Point Enabled Exceptions type of Program interrupt
2. Trace interrupt (for **mtmsr**[**d**] only)

G. Other Instructions
1. Exceptions that are mutually exclusive and the same priority:
   a. Trap type of Program interrupt
   b. System Call
   c. Privileged Instruction type of Program interrupt
   d. Illegal Instruction type of Program interrupt
2. Trace interrupt

H. Instruction Segment Interrupt

I. Instruction Storage Interrupt

IV. Thermal Management Interrupt

V. System Error Interrupt

VI. Maintenance Interrupt

VII. External Interrupt

A. Direct

B. Mediated

VIII. Hypervisor Decrementer Interrupt

XI. Decrementer Interrupt

9.11 Interrupt Latencies

Latencies for taking various interrupts are variable, based on the state of the machine when conditions exist for an interrupt to be taken. Instruction dispatch is blocked for both threads when the conditions exist for a system-caused exception to be taken. Since microcoded instructions must complete, the interrupt latency depends on the current instructions being processed. The interrupt is not taken until both threads are flushed, no higher-priority exceptions exist, and the exception condition, if enabled, exists.

9.12 Machine State Register Settings Due to Interrupts

Table 9-40 shows the settings of Machine State Register (MSR) bits for Hypervisor (HV), Machine-Check Enable (ME), and Recoverable Interrupt (RI), relative to the interrupt type. *Table 9-3* on page 244 list the interrupts supported by the PPE, the effective address of the interrupt handler (interrupt vector), and summarizes the conditions that cause the interrupt.

Table 9-40. Machine State Register (MSR) Bit-Settings Due to Interrupts

Interrupt Type	MSR Bit[1]		
	HV	ME	RI[2]
System Reset	1	—	0
Machine Check (imprecise form)	1	0	0[3]
Precise Machine Check	1	0	0[3]
Data Storage	m	—	0
Data Segment	m	—	0
Instruction Storage	m	—	0
Instruction Segment	m	—	0
External	e	—	0[4]
Alignment	m	—	0
Program	m	—	0
Floating-Point Unavailable	m	—	0
Decrementer	m	—	0
Hypervisor Decrementer	1	—	—
System Call	s	—	0
Trace	m	—	0
System Error	1	—	—
Maintenance	1	—	—
Thermal Management	1	—	—
VXU Unavailable	m	—	0

Legend:
- 0 Bit is set to ‘0’.
- 1 Bit is set to 1.
- — Bit is not altered
- m MSR[HV] is set to ‘1’ if LPCR[LPES] bit 0 = ‘0’ and LPCR[LPES] bit 1 = ‘0’; otherwise, the state of MSR[HV] is not altered.
- e MSR[HV] is set to ‘1’ if LPCR[LPES] bit 0 = ‘0’; otherwise, the state of MSR[HV] is not altered.
- s MSR[HV] is set to ‘1’ if LEV = ‘1’ or LPCR[LPES] bit 0 = ‘0’ and LPCR[LPES] bit 1= ‘0’; otherwise, the state of MSR[HV] is not altered.

Note:

1. MSR[BE], MSR[EE], MSR[FE0], MSR[FE1], MSR[FP], MSR[PR], MSR[SE], MSR[DR], and MSR[IR] are cleared to ‘0’, and MSR[SF] is set to ‘1’.
2. MSR[RI] is not changed for interrupts that use HSRR0 or HSRR1.
3. MSR[RI] is cleared to ‘0’. SRR1[62] is cleared to ‘0’ for an imprecise Machine Check and is not altered for a precise Machine Check.
4. MSR[RI] is not altered if LPCR[LPES] bit 0 is set to ‘0’ and HID0[extr_hsrr] is set to ‘1’; otherwise, MSR[RI] is cleared to ‘0’.

9.13 Interrupts and Hypervisor

The Hypervisor feature supported by the PPE is described in *Section 11* on page 327. This section and *Table 9-40* on page 290 contain details relating to the requirements for and effects of interrupts in Hypervisor mode.

The main setup for interrupt handling is to place code in the interrupt vector offset. Each logical partition (`LPAR`) can have its own interrupt vector using the Real Mode Offset Register (`RMOR`). This register points to the start of the interrupt vector (defaults to 0), so that each `LPAR`—by means of a hypervisor (*Section 11* on page 327)—can have its own dedicated vectors. There is also a hypervisor-only interrupt vector offset register called `HRMOR` which points to the interrupt vector when `MSR[HV]` = '1'.

The following areas need to be initialized:

- `HRMOR` for setting Hypervisor interrupt vector area.
- `RMOR` for pointing to the current supervisor's interrupt vector area (which should change for each `LPAR`).
- Each enable bit (and any mask bits) listed in *Table 10-9* on page 324.
- External Interrupt settings.
- Interrupt priority settings by means of `IIC_CPL0` and `IIC_CPL1` (internal interrupt controller current priority level for thread 0 and thread 1). Default is 0, which is the highest priority.

9.14 Interrupts and Multithreading

The multithreading features supported by the PPE are described in *Section 10* on page 295. For details on the entry condition, activated thread, mask condition, and suspended-thread behavior for each interrupt type, see *Section 10.8.4 Thread Targets and Behavior for Interrupts* on page 324.

9.15 Checkstop

Checkstop is a special non-recoverable condition (rather than an interrupt) in which the PPE and the rest of the CBE processor shut down. There are multiple causes for Checkstop. It can be caused, for example, by two successive Machine Check interrupts occurring close together, with the second occurring before the first one completes. It can also be caused by a watchdog timeout or a hardware signal from an external device.

Software cannot accept nor mask a Checkstop event. After the CBE hardware initiates the Checkstop sequence, software has no role until System Reset occurs. See the *Cell Broadband Engine Hardware Initialization Guide* for more information about Checkstop.

9.16 Use of an External Interrupt Controller

The primary purpose of the IIC is to allow interrupts from SPEs and other components in the CBE processor to be handled internally, without using a external interrupt controller. In a system with many I/O devices, however, an external interrupt controller may be desirable. In such a configu-

ration, the IICs can function as second-level interrupt controllers, handling all interrupts internal to a CBE processor or within a multi-CBE-processor system, while an external system interrupt controller handles interrupts external to the CBE processor or multi-CBE-processor system.

When handling an interrupt in a system with an external main system interrupt controller, PPE software must first check the IICs to determine if the interrupt was sourced internally or from an external system interrupt controller.

9.17 Relationship Between CBE and PowerPC Interrupts

The following differences exist between CBE and PowerPC Architecture interrupts:

- PowerPC Architecture interrupts not supported by the CBE processor:
  - Imprecise-Mode Floating-Point Enabled.
  - Performance Monitor. (But see the unrelated CPE Performance Monitor interrupt, below.)
  - Optional Example Extensions to the Trace Facility.
- CBE Interrupts supported differently than in the PowerPC Architecture:
  - In the PowerPC Architecture, the Machine Check interrupt is implementation-dependent. In the CBE, the interrupt is supported in two forms, *precise*[21] and *imprecise*, neither of which are defined in the PowerPC Architecture.
  - In the PowerPC Architecture, External interrupts have a single form. In the CBE, the interrupts are supported in two forms, *direct* and *mediated*, neither of which are defined in the PowerPC Architecture.
- CBE Interrupts not defined in the *PowerPC Architecture* or the Vector/SIMD Multimedia Extension to the *PowerPC Architecture*:
  - System Error interrupt.
  - Maintenance interrupt (instruction-caused).
  - Maintenance interrupt (system-caused).
  - Thermal Management interrupt.
  - External Interrupts:
    - SPU Instruction interrupt.
    - MFC Command interrupt.
    - MIC Auxiliary Trace Buffer Full interrupt.
    - EIB Possible Livelock Detection interrupt.
    - Token Manager interrupt.
    - CPE Performance Monitor interrupt (unrelated to PowerPC Performance Monitor interrupt).
    - Software interrupt.

21. The precise Machine Check interrupt is a system-caused interrupt but appears to be an instruction-caused interrupt from the viewpoint of the PPE thread that causes it (see *Section 9.5.2* on page 246).

10. PPE Multithreading

The CBE processor elements use several implementation-specific techniques to speed program execution. In the PPE, simultaneous multithreading is one of the most important techniques. To software, the PPE appears to provide two independent instruction-processing units. The threads appear to be independent because the PPE provides each thread with a copy of architectural state, such as general-purpose registers, but the threads are not completely independent because many execution resources are shared by the threads to reduce the hardware cost of multithreading.

To software, the PPE implementation of multithreading looks similar to a multiprocessor implementation, but there are several important differences. *Table 10-1* compares the PPE multithreading implementation to a conventional dual-core microprocessor.

Table 10-1. PPE Multithreading versus Multi-Core Implementations

Characteristic	PPE Multithreading	Multiple Cores on a Single Chip
Hardware cost	Low to moderate cost; the processor replicates only architectural state.	High cost; the entire processor core and caches are replicated.
Performance/Cost	A typical 10% to 30% performance gain (application-dependent) for 5% hardware cost.	Up to 100% performance gain for 100% hardware cost.
Hardware efficiency	Execution-unit utilization increased compared to single-thread of execution.	Execution-unit utilization unchanged.
Software restrictions	Threads must execute in same logical partition (must share memory-management tables, etc.).	Depending on implementation, threads and processes may be able to execute in different logical partitions.
Thread competition	Threads may compete for execution resources, such as cache residency and execution units.	Threads do not compete; they have private, replicated execution resources.
System throughput	System throughput improved at possible expense of slightly slower single-thread latency.	System throughput improved without reducing single-thread latency.

Since most of the PPE hardware is shared by the two threads of execution, the hardware cost of the PPE multithreading implementation is dramatically lower than the cost of replicating the entire processor core. The PPE's dual-threading typically yields about one-fifth the performance increase of a dual-core implementation, but the PPE achieves this 10%-to-30% performance boost at only one-twentieth of the cost of a dual-core implementation.

10.1 Multithreading Guidelines

In the PPE multithreading implementation, the two hardware threads share execution resources. Consequently, concurrent threads that tend to use different subsets of the shared resources will best exploit the PPE's multithreading implementation. Pointer chasing and scattered memory accesses are an excellent examples of applications in which multithreading really shines, even if both threads are running the same type of memory-latency-sensitive application. For example, a speedup of 2x is entirely possible when running memory-pointer-chases on both threads.

When one thread experiences high miss rates in the L1 and/or L2 caches, the other execution resources in the PPE, such as the fixed- and floating-point units, will be completely free to execute instructions from the other thread. Two examples of typical software functions that are likely to incur cache misses on memory references are the process of gathering data from sparse or scattered vectors in preparation for SIMD processing and the pointer-chasing performed when searching a linked list. For example, searching a linked list for an element of a particular type might be done with a loop like the following:

```
ptr = list_head;
while (ptr != NULL)
{
        if (ptr->type == desired_type) {
            break;
        }
        ptr = ptr->next;
}
```

The code above accesses list elements in link order, but the physical-memory addresses of adjacent list elements may not be adjacent. Consequently, on each iteration of the loop above, every execution of the statement `ptr = ptr->next` is likely to cause a cache miss the first time through the loop. If the list is large enough to overwhelm the data cache(s), it may cause cache misses every time a link is accessed.

Floating-point operations also often leave most of the other execution resources sitting idle. When one thread makes heavy use of the floating-point execution unit, a second, fixed-point-intensive thread will be able to execute freely and make significant forward progress when the pipeline would otherwise be idle.

Certain types of source code result in instruction sequences that are dominated by serial dependencies (each successive instruction depends on the previous one) and so have little instruction-level parallelism. Similarly, when compute-intensive applications have loops that cannot be effectively unrolled or that prevent successful software pipelining, the resulting thread of execution will not be able to exploit fully the available parallelism in the PPE execution pipeline. In these situations, when a single thread lacks enough instruction-level parallelism, a second simultaneous thread can help increase the use of execution resources and therefore system throughput.

Another specific situation that leaves execution resources idle is a thread that has a lot of branches that cannot be successfully predicted by the PPE branch-prediction hardware. A mispredicted branch leaves the execution resources idle for many cycles. A second, more-predictable thread would be able to make good forward progress and increase pipeline use and system throughput.

Whenever the overriding goal for a CBE application is to maximize system throughput, multithreading should be used. When the overriding goal is to minimize the latency of a particular task, the system should be programmed to disable the second thread or to set its priority low enough to reduce competition for execution resources. However, setting the second thread's priority low will not eliminate latency-inducing effects on the other thread. If the target is minimum *guaranteed* latency, the processor should ideally be in single-thread mode or should not have a second thread running at all (that is, the thread context is sleeping), because some operations on the low-priority thread—for example, a **sync** operation—can still cause the entire pipeline to stop.

PPE multithreading will not usually help code that is tuned precisely to the characteristics of the pipeline, because such a thread can probably use all or almost all of the execution resources. Even in this case, using multithreading will not necessarily compromise system throughput unless performance of the tuned thread also depends on the sizes of the L1 and/or L2 caches; a thread that is tuned to the cache sizes can experience thrashing if a second thread is allowed to run and pollute the cache. Cache misses are so expensive that any increase in misses will probably more than negate the benefit gained from multithreading[22].

10.2 Thread Resources

For each thread, the PPE has duplicate copies of all architected state, such as general-purpose registers and thread-dependent Special-Purpose Registers (SPRs). Exceptions to this rule are registers that deal with system-level resources, such as logical partitions and memory management. To software, the PPE appears to offer two separate virtual processors for instruction execution.

Resources that are not described in the *PowerPC Architecture*, such as the hardware execution units, are usually shared by the two threads, but non-exposed resources are duplicated in cases where the resource is small or offers a performance improvement that is critical to multithreaded applications.

The two PPE threads are independent in most aspects, but they share logical-partitioning resources. Thus, the threads always execute in the same logical-partition context and share structures such as virtual-memory-mapping tables. Threads also share most of the large arrays and queues, such as caches, that consume significant amounts of chip area.

10.2.1 Registers

The following architected registers are *duplicated* for multithreading and used by software running in any privilege state (including problem-state):

- General-Purpose Registers (GPRs) (32 entries per thread).
- Floating-Point Unit Registers (FPRs) (32 entries per thread).
- Vector Multimedia Registers (VMRs) (32 entries per thread).
- Condition Register (`CR`).
- Count Register (`CTR`).
- Link Register (`LR`).
- Fixed-Point Exception Register (`XER`).
- Floating-Point Status and Control Register (`FPSCR`).
- Vector Status and Control Register (`VSCR`).
- Decrementer (`DEC`).

22. Multithreading may result in fewer cache misses if both threads use the same code, as for example when using shared library routines.

In addition, see *Section 10.4 Thread Control and Status Registers* on page 302 for additional architected registers that are duplicated for multithreading but used specifically for thread control and status monitoring. All Memory Mapped I/O (MMIO) registers in the PPE deal with aspects of the Power Processor Storage System (PPSS) that are shared for both threads, so all MMIO registers are shared for both threads as well.

10.2.2 Arrays, Queues, and Other Structures

The following arrays, queues, and structures are fully *shared* between threads running in any privilege state:

- L1 Instruction Cache (ICache), L1 Data Cache (DCache), and L2 Cache.
- Instruction and Data Effective-to-Real-Address Translation tables (I-ERAT and D-ERAT).
- I-ERAT and D-ERAT Miss Queues.
- Translation Lookaside Buffer (TLB).
- Load Miss Queue and Store Queue.
- Microcode Engine.
- Instruction Fetch Control.
- All execution units:
  - Branch (BRU).
  - Fixed-Point Integer Unit (FXU).
  - Load and Store Unit (LSU).
  - Floating-Point Unit (FPU).
  - Vector Media Extension Unit (VXU).

The following arrays and queues are *duplicated* for each thread.

- Segment Lookaside Buffer (SLB).
- Branch History Table (BHT), with global branch history.
- Instruction Buffer (IBuf) Queue.
- Link Stack Queue.

Duplicating the instruction buffer allows each thread to dispatch regardless of any dispatch stall in the other thread. Duplicating the SLB is convenient for the implementation because of the nature of the *PowerPC Architecture* instructions that access it and because it is a relatively small array.

The instruction-fetch control is shared by both threads because the instruction cache has only one read port and so fetching must alternate between threads every cycle. Each thread maintains its own BHT and GBH to allow independent and simultaneous branch prediction.

10.2.3 Pipeline Sharing and Support for Multithreading

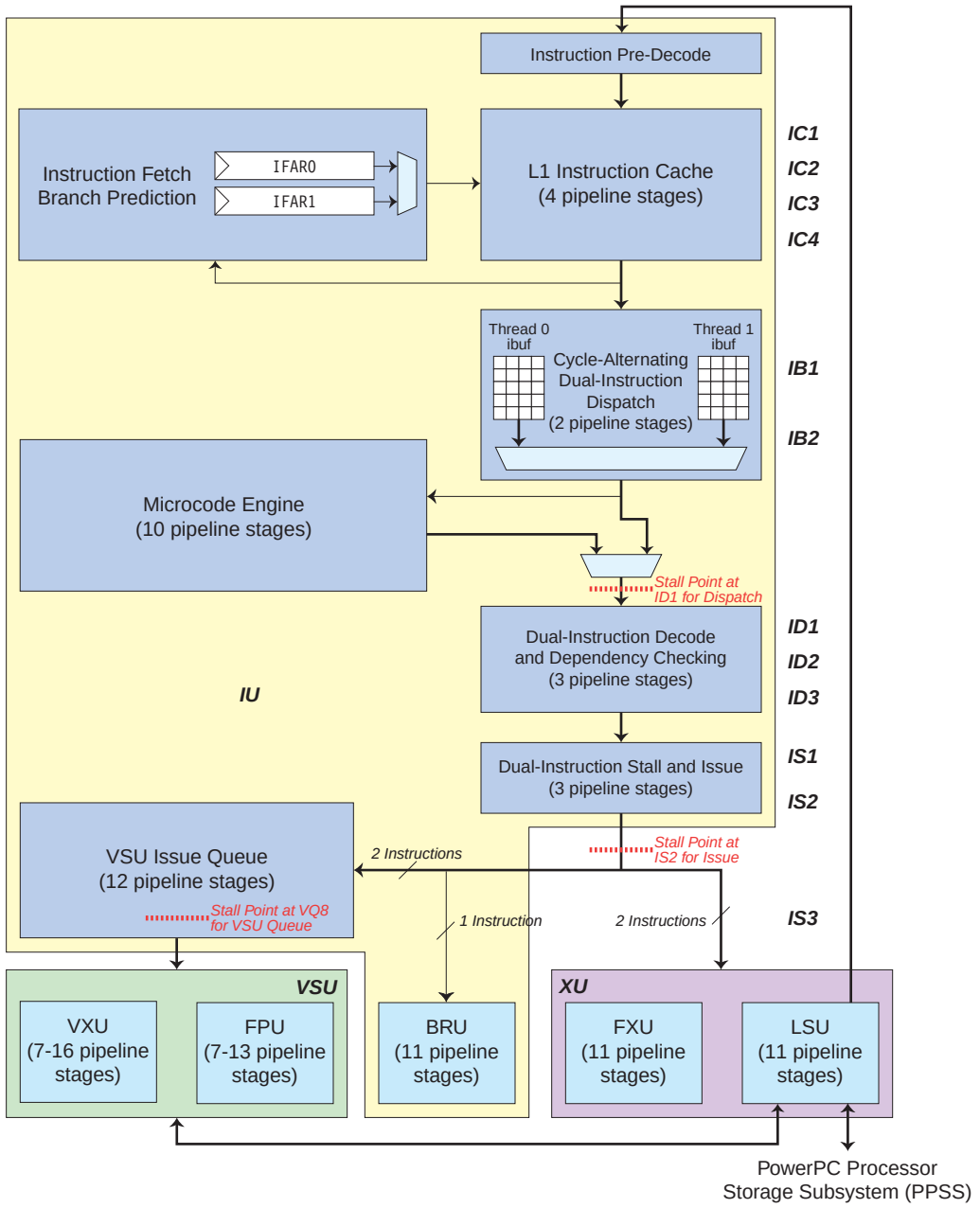
Figure 10-1 on page 300 shows a high-level, multithreading view of instruction flow and resources in the PPE pipeline. The units shown in this figure include the Instruction Unit (IU), Execution Unit (XU), Fixed-Point Unit (FXU), Load-Store Unit (LSU), Branch Unit (BRU), Vector

Scalar Unit (VSU), Floating-Point Unit (FPU), and Vector/SIMD Multimedia Extension Unit (VXU). The pipeline stages shown for the execution units at the bottom of this figure are the total number of stages through completion; bypass paths are reached in fewer stages. For details on actual latency and throughput for each instruction, see *Table A-1 PowerPC Instructions, by Execution Unit* on page 712.

Instruction-fetch hardware autonomously operates and maintains an Instruction-Fetch Address Register for each thread (`IFAR0` and `IFAR1`), which are not visible to software. In contrast to instruction dispatch, which can repeatedly favor one thread over another, instruction fetching strictly alternates on successive cycles. The purpose of instruction fetching is to keep each thread's instruction buffer full with useful instructions that are likely to be needed by the thread. A thread's `IFAR` register is distinct from its program counter: the program counter tracks *actual* instruction-execution flow while an IFAR tracks *predicted* instruction-execution flow. Instruction fetching is pipelined; starting the fetch of a new sequential stream after a predicted-taken branch requires eight cycles.

The dispatch hardware sends instructions from the instruction buffers to the shared decode, dependency-checking, and issue pipeline. The dispatch hardware selects instructions from the buffer of a particular thread based on thread priority and other factors such as stall conditions. Each pipeline stage beyond the dispatch point contains instructions from one thread only.

Figure 10-1. PPE Multithreading Pipeline Flow and Resources



10.2.3.1 *Pipeline Stall Points*

There are three stall points in the pipeline: dispatch (ID1), issue (IS2), and VSU issue (VQ8). The dispatch stall point at ID1 (*Figure 10-1*) is separate for each thread so that one thread may dispatch even if the other is stalled.

The issue and VSU-issue stall points (see *Appendix A.5 Issue Rules* on page 750 and *Appendix A.6.2 Vector/Scalar Unit Issue Queue* on page 754) are not separate for each thread. A stall at IS2 requires both threads to stall, and a stall at VQ8 will stall both threads if and only if the stall also results in a stall at IS2. To reduce the number of VQ8 stalls that also stall at IS2, the VSU Issue Queue has a two-stage buffer that can expand the pipeline length temporarily by two stages. When VQ8 stalls and the two-stage buffer is free, the VSU Issue Queue can accept up to four more instructions (2 entries * 2 instructions) before it must signal IS2 to stall.

Most stalls do not cause flushes, refetches, or dispatch blocking. Almost all stalls just cause the stall to occur such that issue is stopped for a few cycles (see *Section 10.7* on page 318). However, for very long-latency conditions, such as when an instruction is issued that is dependant on an L1 cache miss or on a divide that is being computed, stall conditions may cause the following sequence of actions in the pipeline:

- Instructions younger than the stalled instruction are flushed from the pipeline.
- Instructions starting with the stalled instruction are refetched.
- The thread is stalled at the dispatch (ID1) stage until the stall condition is removed.

10.3 Thread States

The operating state of each thread has several properties that affect instruction dispatch, including:

- The thread's privilege state.
- Whether the thread is suspended or enabled.
- Whether or not the thread is blocked by a stall condition not related to its priority.
- The thread's priority.

10.3.1 Privilege States

A thread can run in any of three privilege states: hypervisor state, privileged state (supervisor), and problem (user) state. Hypervisor state is the most privileged, and some system operations require the initiating thread to be in hypervisor state. Hypervisor state exists to run a meta-operating system that manages logical partitions in which multiple operating system instances can run. Supervisor state is the state in which an operating system instance is intended to run when multiple instances are supported. User state (problem state) is for running application programs. *Table 10-2* summarizes the Machine State Register bit settings needed for each state.

Table 10-2. Privilege States

MSR[HV]	MSR[PR]	Privilege State
1	0	Hypervisor
0	0	Privileged State (also called Supervisor)
Don't care	1	Problem State (also called User)

As shown in the table, when MSR[PR] is set to '1', the user state (problem state) is active, regardless of the state of the hypervisor bit in the MSR register.

After POR, the Problem State Change Thread Priority Enable bit, (TSCR[UCP]) and the Privileged But Not Hypervisor State Change Thread Priority Enable bit (TSCR[PSCTP]) are cleared to ‘0’, so thread priority can be changed only in hypervisor state. See *Section 10.4.6* on page 307.

10.3.2 Suspended or Enabled State

A thread can be in the suspended state if its Thread Enable bit is cleared to ‘0’ in the Control Register (CTRL[TE0] or CTRL[TE1]). A thread can suspend only itself (a thread cannot suspend the other thread), and a thread can only write to the CTRL register to change a thread-enable bit when the thread is in hypervisor state (MSR[PR,HV] = 0,1).

10.3.3 Blocked or Stalled State

In general, *blocking* occurs at the instruction-dispatch stage and stops only one of the two threads, whereas *stalling* occur at the instruction-issue stage and stop both threads. A thread can be blocked or stalled for any one of several conditions, including the following:

- The thread executes one of a set of special **nop** instructions that block instruction dispatch for a specific number of cycles (see *Table 10-4* on page 310).
- The pipeline blocks the thread at instruction dispatch because the combination of thread priorities puts in force an instruction-dispatch policy that selects the other thread for dispatch (see *Section 10.5* on page 309).
- The pipeline is forced to stall instruction progress due to dependencies.
- A context-synchronizing event or instruction occurs.
- The HID0[issue_serialize] bit is set to ‘1’.

10.4 Thread Control and Status Registers

Section 10.2.1 on page 297 lists the architected registers that are duplicated, per thread, and used by software running in any privilege state (including problem state). This section describes the primary control and status registers that affect and monitor the behavior of the multithreading facilities in the PPE. These registers allow software to enable threads, set relative thread priorities, set the duty cycle for threads of unequal priority, determine thread response to interrupts, and set timer values so that system software will be informed when a thread is starved of execution resources.

Each thread is viewed as an independent processor, complete with separate exceptions and interrupt handling. The threads can generate exceptions simultaneously, and the PPE supports concurrent handling of interrupts on both threads by duplicating some registers defined by the PowerPC Architecture.

The following registers associated with exceptions and interrupt handling are duplicated or are thread-dependent:

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

- Data Storage Interrupt Status Register (DSISR).
- Decrementer (DEC).
- Logical Partition Control Register (LPCR).
- Data Address Register (DAR).
- Data Address Breakpoint Register (DABR and DABRX).
- Address Compare Control Register (ACCR).
- Thread Status Register Local (TSRL).
- Thread Status Register Remote (TSRR).

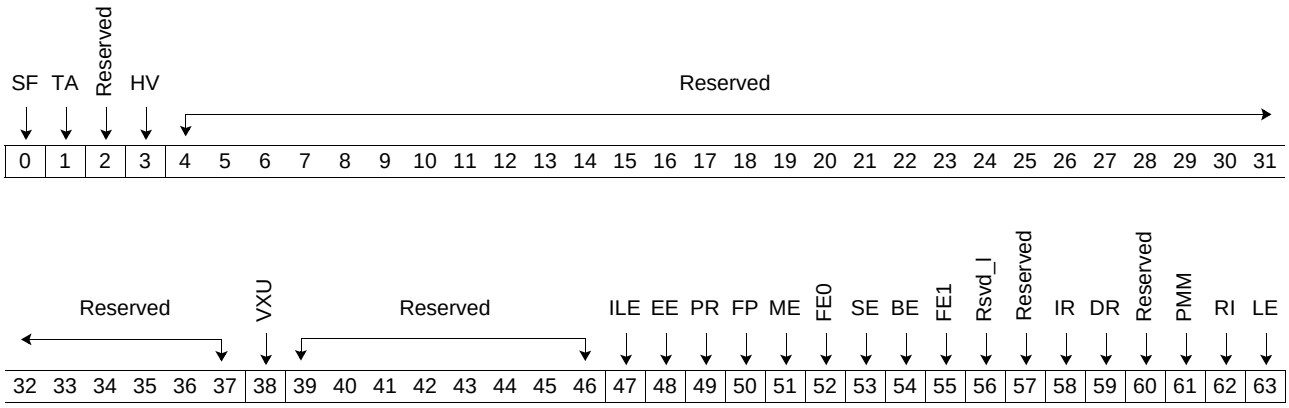
In addition, the following thread-independent registers also are associated with exceptions and interrupt handling on both threads:

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

The following sections describe processor registers that play a central role in controlling and monitoring multithreading activity. The tables that describe register fields have been edited for brevity and clarity (for example, the reserved fields and fields not relevant to multithreading are not described). For complete register descriptions, see the *Cell Broadband Engine Registers* specification.

10.4.1 Machine State Register (MSR)

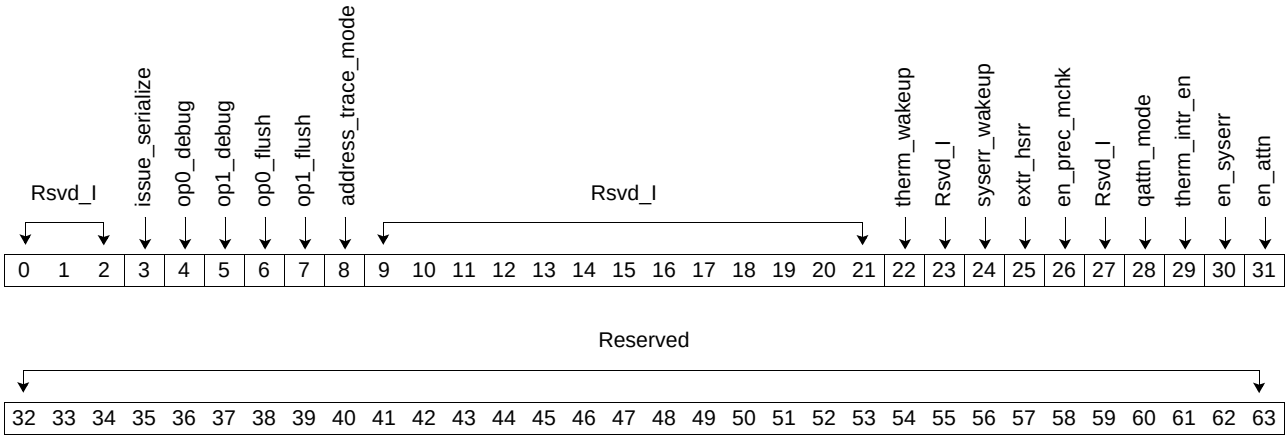
The Machine State Register (MSR) contains the hypervisor-state and problem-state (user-state) bits for setting the privilege state of a thread. See *Section 9.5.7* on page 251 for how external interrupts are masked when MSR[EE] is cleared to ‘0’. When MSR[ME] is set to ‘1’, the thread can take a machine-check interrupt. See *Section 11.2.1* on page 331 for how the MSR[HV] bit is set to enable hypervisor state.



Bit(s)	Field Name	Description
3	HV	Hypervisor State (see *Table 10-2* on page 301)
48	EE	External interrupt enable
49	PR	Problem state (see *Table 10-2* on page 301)
51	ME	Machine check enable

10.4.2 Hardware Implementation Register 0 (HID0)

The Hardware Implementation Register 0 (HID0) holds several bits that determine whether a thread will take an interrupt in response to some exceptions and a bit to set issue-serialize mode. Issue-serialize mode, which is enabled with the issue_serialize bit is set to '1', causes the processor to wait until all previous instructions have completed before issuing the next instruction; this mode also prevents dual instruction issue.

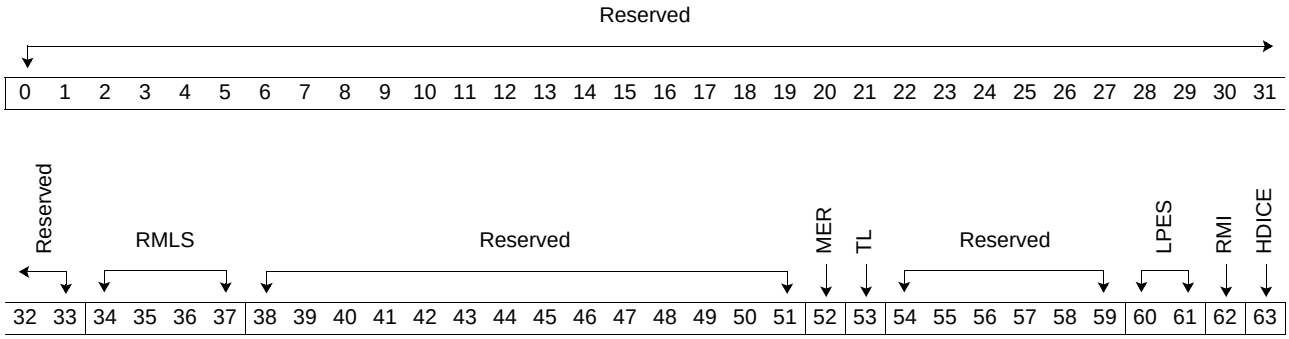


Bit(s)	Field Name	Description
3	issue_serialize	Issue Serialize mode 0 Normal operation 1 Next instruction is not issued until all previous instructions have completed (no dual-issue either).
22	therm_wakeup	Enable thermal management interrupt to wakeup suspended thread **Note:** Wakeup occurs even if HID0[therm_intr_en] = '0'.
24	syserr_wakeup	Enable system error interrupt to wakeup suspended thread Allows the system error interrupt to wake up either thread if it is dormant. When a system error interrupt is received, if this bit is enabled the interrupt wakes up the thread that the interrupt was for (could be both threads). For example. if thread 0 is dormant, thread 1 is active, syserr_wakeup is set, and the interrupt is for thread 0, then thread 0 is awakened (the active thread (thread1) is unaffected since it is already awake). If both threads are dormant and if syserr_wakeup is set, the interrupt awakens both threads. **Note:** Wakeup occurs even if HID0[en_syserr] = '0'.

Bit(s)	Field Name	Description
26	en_prec_mchk	Enable precise Machine Check **Note:** All loads to caching-inhibited (I = '1') space cause the instructions after the load to be flushed, and the thread to be blocked, at dispatch until data is returned for the load.
29	therm_intr_en	Master thermal management interrupt enable Clearing this bit disables all thermal management interrupts regardless of MSR state. 0 Disables all thermal management interrupts regardless of MSR state 1 Enabled
30	en_syserr	Enable system errors generated from outside the PPE. 0 Disabled 1 Enabled

10.4.3 Logical Partition Control Register (LPCR)

The Logical Partition Control Register (LPCR) holds several partition-related processor control bits. LPCR is a partially shared register: the LPCR[RMLS], LPCR[TL], and LPCR[LPES] fields are shared between threads; the LPCR[MER], LPCR[RMI], and LPCR[HDICE] fields are duplicated per thread.



Bit(s)	Field Name	Shared or Duplicated	Description
34:37	RMLS	Shared by all threads	Real mode limit selector
52	MER	Duplicated for each thread	Mediate external exception request (interrupt enable)
53	TL	Shared by all threads	TLB load 0 TLB loaded by processor 1 TLB loaded by software
60:61	LPES	Shared by all threads	Logical partitioning (environment selector)
62	RMI	Duplicated for each thread	Real-mode caching (caching inhibited)
63	HDICE	Duplicated for each thread	Hypervisor decrementer interrupt control enable

10.4.4 Control Register (CTRL)

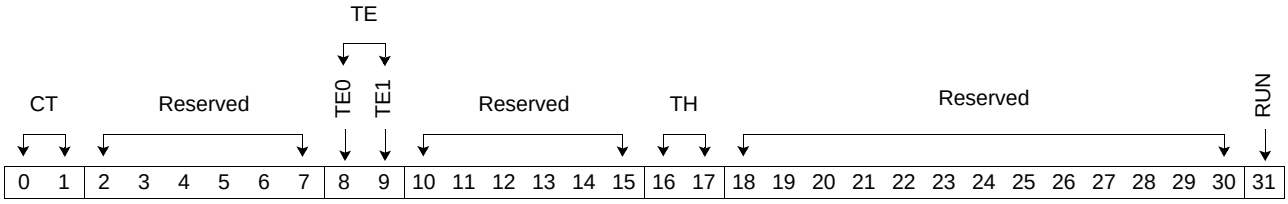
The Control Register (CTRL) holds basic thread status and control bits. Software can use the TE0 and TE1 bits to suspend and resume threads. Hypervisor software executing in one thread can:

- Suspend the present thread.
- Resume the other thread, but cannot suspend the other thread.

Table 10-3 shows the effect on thread state when thread 1 or thread 0 sets the TE field. In this table, "—" means the setting has no effect.

Table 10-3. Effect of Setting the CTRL[TE0] and CTRL[TE1] Bits

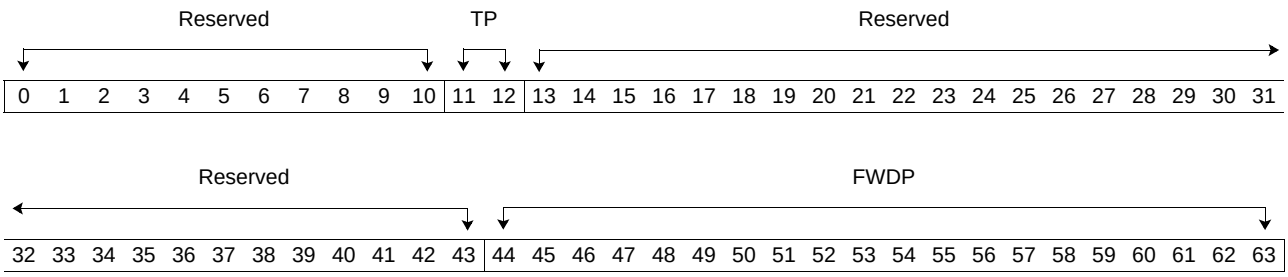
CTRL[TE0]	CTRL[TE1]	If thread 0 sets TE bits		If thread 1 sets TE bits	
		Thread 0	Thread 1	Thread 0	Thread 1
0	0	Suspend	—	—	Suspend
0	1	Suspend	Resume	—	—
1	0	—	—	Resume	Suspend
1	1	—	Resume	Resume	—



Bit(s)	Field Name	Description
0:1	CT	Current thread active (Read Only) These read-only bits contain the current thread bits for threads 0 and 1. Software can read these bits to determine which thread they are operating on. Only one current thread bit is set at a time. 00 Reserved 01 Thread 1 is reading CTRL 10 Thread 0 is reading CTRL 11 Reserved
8:9	TE	Thread enable bits (Read/Write); see *Table 10-3* on page 306. **Note:** Software should not disable a thread when in trace mode (MSR[SE] or MSR[BE] set to '1'). Doing so causes SRR0 to be undefined and can cause a system livelock hang condition.
16:17	TH	Thread history (Read only) If thread A writes CTRL[RUN] then CTRL[16] is set; otherwise if thread B writes CTRL[31] then CTRL[17] is set. These bits cannot be set directly by writing bits 16 or 17 with a **mtctrl** instruction. They are only set when a thread writes CTRL[RUN].
31	RUN	Run state bit.

10.4.5 Thread Status Register Local and Remote (TSRL and TSRR)

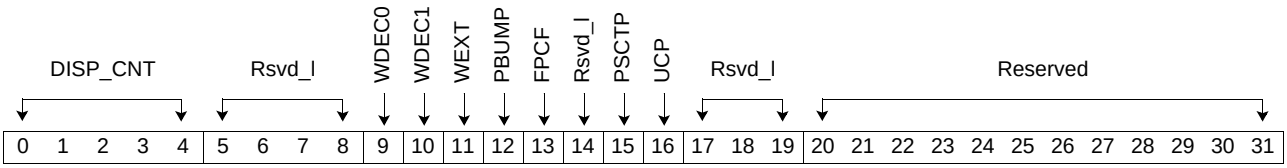
Each thread has a Thread Status Register Local (TSRL) and a Thread Status Register Remote (TSRR). For each thread, the TSRR is a read-only copy of the other thread's TSRL. A thread uses its TSRL to set its priority; a thread uses the TSRR to read the other thread's priority.



Bit(s)	Field Name	Description
11:12	TP	Thread priority (read/write); see *Table 10-6* on page 316. 00 Disabled 01 Low 10 Medium 11 High
44:63	FWDP	Forward progress timer (Read Only); see *Section 10.6.3* on page 317.

10.4.6 Thread Switch Control Register (TSCR)

The Thread Switch Control Register (TSCR) has bits to control the thread's response to interrupts, determine its duty cycle in multithreading mode, and so on.

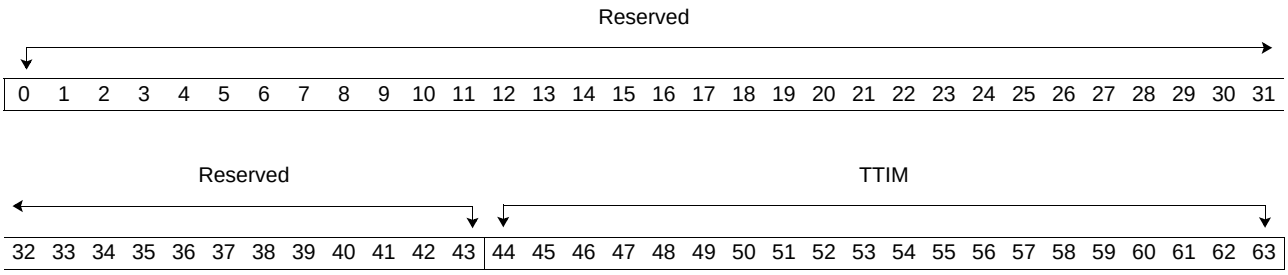


Bit(s)	Field Name	Description
0:4	DISP_CNT	Thread dispatch count; see *Section 10.5* on page 309.
9	WDEC0	Decrementer wakeup enable for thread 0 0 Disabled 1 If a decrementer exception exists, and the corresponding thread is suspended, then the thread is activated.

Bit(s)	Field Name	Description
10	WDEC1	Decrementer wakeup enable for thread 1 0 Disabled 1 If a decrementer exception exists, and the corresponding thread is suspended, then the thread is activated.
11	WEXT	External interrupt wakeup enable 0 Disabled 1 If an external exception exists, and the corresponding thread is suspended, then the thread is activated.
12	PBUMP	Thread priority boost enable 0 Disabled 1 If a system-caused exception is presented, the corresponding interrupt is not masked, and the priority of the corresponding thread is less than medium, sets the priority of the thread to medium. The hardware internally boosts the priority level to medium when the interrupt is pending. This does not change the value in the TSRL[TP] bits for the affected thread. The internal priority remains boosted to medium until an **mttsrl** or priority changing **nop** instruction occurs.
13	FPCF	Forward progress count flush enable This bit only enables or disables the flush from occurring. The forward progress timer does not stop decrementing when set to zero. During normal operation, this bit should be set to '1'. If set to '1', the non-starved thread will be blocked at dispatch until one instruction completes from the starved thread. See TTR[TTIM] and TSRL[FWDP] and *Section 10.6.3* for more information.
15	PSCTP	Privileged but not hypervisor state change thread priority enable; See *Section 10.6.2.1* on page 315. Enables the privileged state (MSR[HV, PR] = '00') to change priority with "**or** Rx, Rx, Rx" **nop** instructions or writes to TSRL[TP]. 0 The ability of the privileged state to change thread priority is determined by TSCR[UCP]. 1 The privileged state can change thread priority to low, medium and high.
16	UCP	Problem state change thread priority enable; see *Section 10.6.2.1* on page 315. Enables the problem state to change priority with "**or** Rx, Rx, Rx" **nop** instructions or writes to TSRL[TP]. 0 The problem state cannot change thread priority. 1 The problem state can change thread priority to low and medium only.

10.4.7 Thread Switch Time-Out Register (TTR)

The Thread Switch Time-Out Register (TTR) has a single field, TTIM. The value in TTIM is copied into the TSRL[FWDP] field every time an instruction from the corresponding thread completes. Thus, with each instruction completion, the forward-progress timer is reset.



Bit(s)	Field Name	Description
44:63	TTIM	Thread time-out flush value; see *Section 10.6.3* for more information.

10.5 Thread Priority

Each thread has an instruction-dispatch priority determined by the value of the TSRL[TP] field. This two-bit field can encode four possibilities:

- Thread disabled.
- Low priority.
- Medium priority.
- High priority.

Software, in particular operating-system software, sets thread priorities to cause hardware to allocate instruction-dispatch bandwidth according to the throughput requirements of the programs running in the threads. For example, software may set thread priorities to cause hardware to favor a foreground thread over a background thread when allocating instruction-dispatch slots (see *Table 10-5* on page 311).

10.5.1 Thread-Priority Combinations

Table 10-4 lists all possible combinations of thread priorities and values of TSCR[DISP_CNT] and describes the instruction-dispatch policy that results from each combination. For some combinations of thread priorities, the value in TSCR[DISP_CNT] allows software to set the duty cycles of instruction dispatching for the two threads.

Though the effects of some of the priority combinations are subtle, there are some basic concepts governing the response of hardware to thread priority settings:

- When the threads have equal priorities, hardware uses fair, round-robin scheduling.
- Dispatch is attempted every cycle except when both threads are set at low priority and the TSCR[DISP_CNT] field is not equal to one.
- When the threads have unequal priorities, the lower-priority thread can be prevented from dispatching any instructions for long periods of time when TSCR[DISP_CNT] is equal to one or when one thread is set to high priority and the other is set to low priority. See *Section 10.6.3 Preventing Starvation: Forward-Progress Monitoring* on page 317 for details.

Table 10-4. Relative Thread Priorities and Instruction-Dispatch Policies

TSCR[DISP_CNT]	Thread 0 Priority	Thread 1 Priority	Attempt to Dispatch	Dispatch Policy Description
== 1	Same		Every Cycle	Even priority; a round-robin algorithm is used to choose a thread for dispatch.
== 1	Different		Every Cycle	Uneven priority; the higher-priority thread always gets the dispatch slot if it can use it; otherwise, the lower-priority thread can use the dispatch slot.
!= 1	Low	Low	Once every DISP_CNT cycles	Dispatch attempted once in TSCR[DISP_CNT] cycles; no dispatch is attempted on the other TSCR[DISP_CNT]–1 cycles. Even priority; a round-robin algorithm is used to choose a thread for dispatch; if the chosen thread in a given cycle cannot dispatch for other reasons (for example, instruction buffer empty), the other thread is allowed to dispatch (priority can toggle between threads).
!= 1	Low	Medium	Every Cycle	Out of every TSCR[DISP_CNT] cycles, one dispatch cycle is given to the lower-priority thread. The other TSCR[DISP_CNT]–1 dispatch cycles are given to the higher priority thread.
!= 1	Medium	Low		
!= 1	Medium	High		
!= 1	High	Medium		
!= 1	Low	High	Every Cycle	Out of every TSCR[DISP_CNT] cycles, one dispatch cycle is given to the lower priority thread only if the higher-priority thread cannot dispatch during that cycle. The other TSCR[DISP_CNT]–1 dispatch cycles are given to the higher priority thread.
!= 1	High	Low		
!= 1	Medium	Medium	Every Cycle	Even priority; a round-robin algorithm is used to choose a thread for dispatch. If dispatch is stalled for the chosen thread in a cycle, no dispatch occurs in that cycle (priority does not toggle between threads).
!= 1	High	High		

There are two special cases for the value of TSCR[DISP_CNT]. When the actual value is cleared to ‘0’, hardware uses a dispatch count of 32, the maximum value. When TSCR[DISP_CNT] is set to ‘1’, the higher-priority thread always gets the dispatch slot if it can use it, otherwise the lower-priority thread gets the slot; there is no blocking.

10.5.2 Choosing Useful Thread Priorities

This section discusses thread priorities from a high-level perspective. A full discussion of priorities, including low-level details, is given in *Section 10.5* beginning on page 309 and *Section 10.6.2* beginning on page 315.

Each thread can be given one of three priorities: low, medium, or high. Thread priority is normally determined by the value in the Thread Priority field of the Thread Status Register Local (TSRL[TP]). For certain combinations of thread priorities, the Dispatch Count field in the Thread Switch Control Register (TSCR[DISP_CNT]) sets the duty cycle of instruction dispatch for the threads. See *Section 10.5.3* for examples of setting TSCR[DISP_CNT].

Together with the value of the TSCR[DISP_CNT] field, there are many possible combinations of thread priorities (see *Table 10-4* on page 310). In practice, thread priorities will most often be set to one of three useful combinations, as shown in *Table 10-5*.

Table 10-5. Three Useful Thread Priority Combinations

Purpose	Thread 0 Priority	Thread 1 Priority	TSCR[DISP_CNT]	Comment
Two Foreground Threads (normal operation)	Medium	Medium	Don't care	PPE attempts to dispatch on every cycle; threads alternate dispatch on each attempt. Software can boost a thread to high when minimum latency is important.
One Foreground Thread One Background Thread	Medium	Low	Higher value puts Low thread farther in the background	PPE attempts to dispatch on every cycle; medium-priority thread gets most dispatch attempts; low-priority thread gets one dispatch attempt out of every TSCR[DISP_CNT] cycles.
	Low	Medium		
Two Background Threads (save power)	Low	Low	Higher value decreases power consumption	PPE attempts to dispatch only once every TSCR[DISP_CNT] cycles; threads alternate dispatch on each attempt.

The first combination sets both threads to medium priority. This combination causes the PPE to attempt to use all available dispatch opportunities and use a fair, round-robin scheduling algorithm. When a thread needs a temporary boost in priority to, for example, execute a time-critical interrupt handler, it can assume high priority to cause the PPE to choose it for most dispatch opportunities.

The second combination sets one thread at medium priority and one to low. This combination causes the PPE to attempt to use almost all available dispatch opportunities for the medium-priority thread; thus, this combination is appropriate to run a low-priority program in the background during otherwise normal operation. The background program will get some execution time, but it will minimally hinder the foreground thread's progress.

The third combination sets both threads at low priority. This combination causes the PPE to attempt dispatch only once every TSCR[DISP_CNT] cycles with a fair, round-robin scheduling algorithm. With a high value in TSCR[DISP_CNT], the PPE will be mostly idle, which will reduce power consumption and heat production while keeping the two threads alive and responsive to changes in system conditions.

One combination that would cause problems—and therefore is not explicitly supported—is a high-priority thread plus a low-priority thread. In such a case, the PPE would never dispatch instructions from the low-priority thread unless the cycle were the TSCR[DISP_CNT] cycle and the high-priority thread were unable to dispatch. The low priority thread would thus starve for dispatch opportunities. Because of this, the combination of high-priority plus low-priority is explicitly not supported.

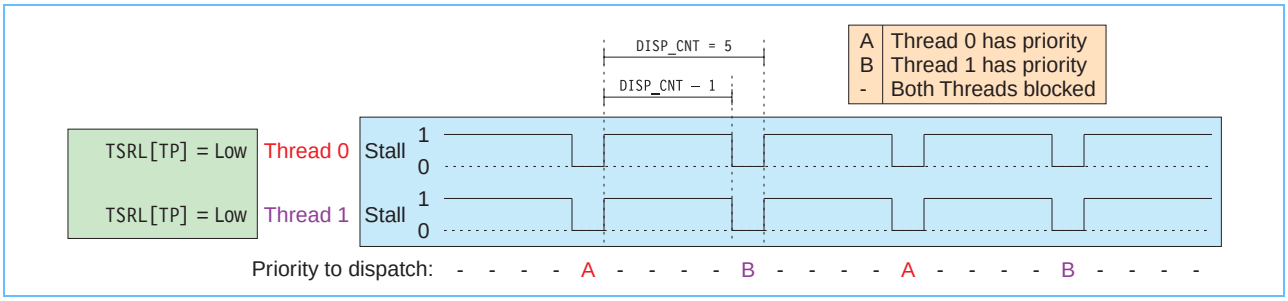
The PPE has a completion-count time-out feature to catch the extreme cases of thread starvation that can result from using this combination. See *Figure 10-4* on page 313 for an illustration of how this combination can cause starvation. See *Section 10.6.1* on page 315 for a discussion of preventing starvation.

10.5.3 Examples of Priority Combinations on Instruction Scheduling

10.5.3.1 *Low Priority Combined with Low Priority*

Figure 10-2 on page 312 illustrates nominal instruction-dispatch activity when both threads are set at low priority. Because the threads have equal priority, the scheduling is fair, with each thread getting an equal chance to dispatch. The result of this combination is that instructions from thread 0 are dispatched only once every ten cycles; similarly, instructions from thread 1 are dispatched only once every ten cycles.

Figure 10-2. Thread Priorities and Instruction Dispatch: Low Priority with Low Priority



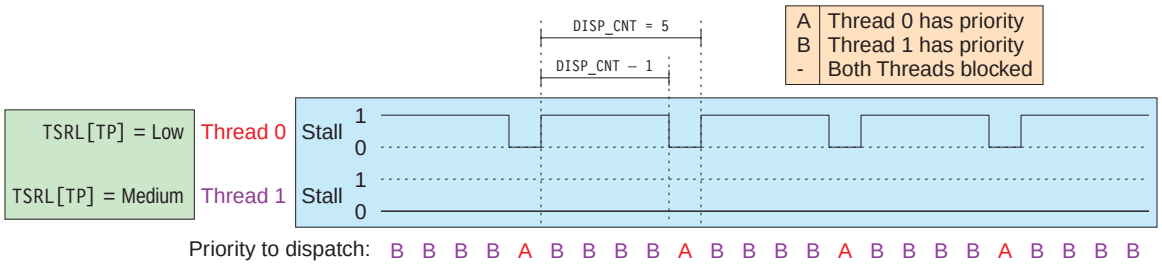
The duty cycle, which is the value in `TSCR[DISP_CNT]`, is set at five. With both threads set at low priority, the processor only dispatches an instruction once every five cycles; for this combination, the duty-cycle value describes the ratio of cycles spent attempting to dispatch to cycles spent idling. A given attempt to dispatch an instruction may not succeed if the threads are blocked for other reasons.

The dispatch stall signal from the priority logic releases the dispatch logic only every fifth cycle, and it releases the stall for both threads. If one thread cannot dispatch for other reasons, the other thread will be allowed to dispatch even if it was selected for dispatch on the previous attempt.

10.5.3.2 *Low Priority Combined with Medium Priority*

Figure 10-3 illustrates nominal instruction-dispatch activity when one thread is set at low priority and the other one at medium. Since the threads have unequal priority, the scheduling favors the higher-priority thread. The result of this combination is that instructions from thread 1 are dispatched on four out of five cycles while instructions from thread 0 are dispatched on only one out of five cycles.

Figure 10-3. Example 2 of Thread Priorities and Instruction Dispatch



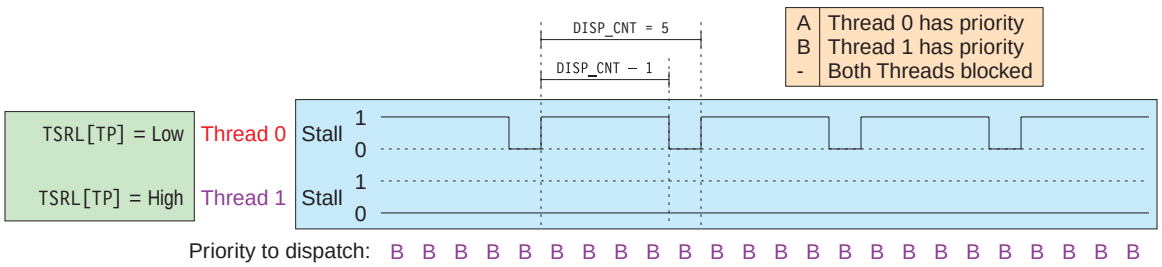
The duty cycle, `TSCR[DISP_CNT]`, is again set at five. With this combination of priorities, the processor dispatches an instruction every cycle, and the duty-cycle value describes the ratio of cycles spent on attempts to dispatch exclusively from thread 1 (the higher-priority thread) to cycles spent giving instructions from thread 0 a chance to dispatch.

The dispatch stall signal from the priority logic for thread 1 is always false; with this combination of priorities, instruction dispatch from thread 1 is always allowable. The dispatch stall signal for thread 0 is false (dispatch allowed) only one cycle in five. On this cycle, priority is given to thread 0 if dispatch is not otherwise stalled. If thread 0 is stalled for other reasons, the dispatch logic will attempt to dispatch from thread 1.

10.5.3.3 ***Low Priority Combined with High Priority***

Figure 10-4 illustrates nominal instruction-dispatch activity when one thread is set at low priority and the other at high. The scheduling favors the higher-priority thread to the exclusion of the low-priority thread; in the segment of activity shown, instructions from thread 0 are never given a opportunity to dispatch.

Figure 10-4. Example 3 of Thread Priorities and Instruction Dispatch



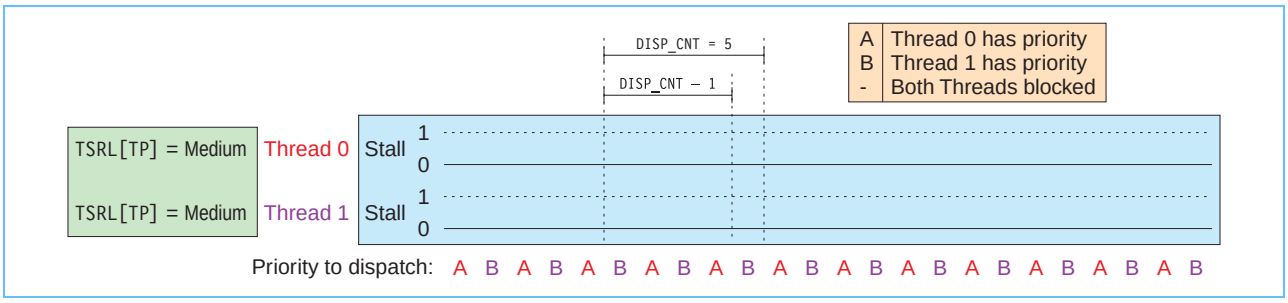
The duty cycle is again set at five. With this combination of priorities, the processor dispatches an instruction every cycle, and the duty-cycle value describes the ratio of cycles spent on attempts to dispatch exclusively from thread 1 to cycles spent giving instructions from thread 0 a chance to dispatch. For this combination, however, dispatch from thread 0 is attempted only if thread 1 is unable to dispatch. Consequently, it is possible that no instructions from thread 0 would ever be dispatched because it is possible that thread 1 will always be ready for dispatch. To keep thread 0 from being completely starved in a scenario like this, the PPE can be programmed to monitor forward-progress and force a dispatch opportunity for thread 0 when starvation exceeds a programmable limit (see *Section 10.6.3* on page 317).

Although the stall signal from the priority logic for thread 0 is false every fifth cycle, no instruction from thread 0 is dispatched because, in this example, thread 1 is never stalled for other reasons on the cycles when thread 0 is considered for dispatch.

10.5.3.4 *Medium Priority Combined with Medium Priority*

Figure 10-5 illustrates nominal instruction-dispatch activity when both threads are set at medium priority; the same dispatch activity applies when both threads are set to high priority. Because the priorities are equal, the scheduling is fair with each thread getting an equal chance for dispatch. The result of this combination of priorities is that the dispatch alternates threads so that a given thread is allowed to dispatch once every other cycle.

Figure 10-5. Example 4 of Thread Priorities and Instruction Dispatch

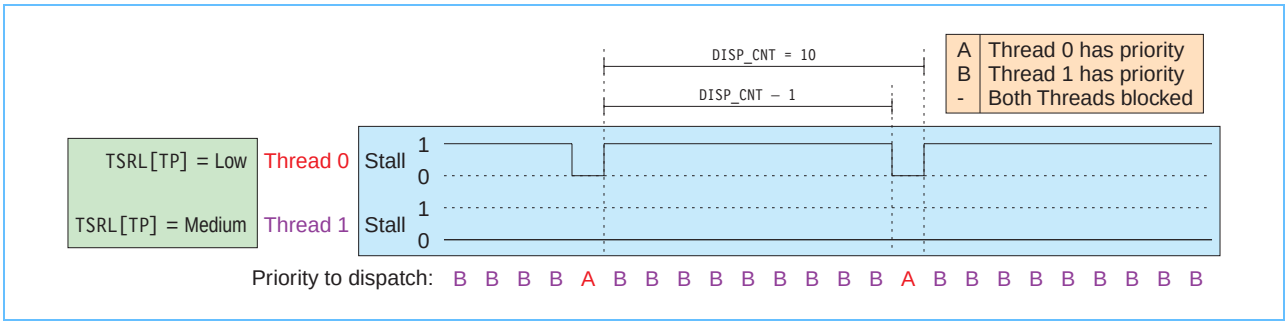


The duty cycle is again five, but with this combination of priorities, the duty cycle value, `TSCR[DISP_CNT]`, plays no part in choosing a thread for dispatch consideration. Dispatch is attempted on every cycle, and the dispatch logic alternates the thread it chooses for consideration. If the thread under consideration in a given cycle is unable to dispatch an instruction, the other thread will be allowed to dispatch even if it was selected for dispatch on the previous attempt.

10.5.3.5 *Low Priority Combined with Medium Priority and Larger `TSCR[DISP_CNT]`*

Figure 10-6 illustrates nominal instruction-dispatch activity when one thread is set at low priority and the other at medium. The situation and resulting dispatch behavior is identical to that explained in the second example above in *Section 10.5.3.2* on page 312, but the duty-cycle value is set at ten, which causes the dispatch logic to give an even greater preference to instructions from thread 1. The ratio is nine-to-one instead of four-to-one as in the previous example.

Figure 10-6. Example 5 of Thread Priorities and Instruction Dispatch



10.6 Thread Control and Configuration

Software can configure and control thread behavior in several ways, including:

- Enabling and suspending a thread.
- Setting the instruction-dispatch policy by setting thread priority.
- Executing a special **nop** instruction to cause temporary dispatch stalls.
- Setting an appropriate forward-progress timer value to allow hardware to guarantee a minimum amount of execution time.
- Switching the state of the processor between single-threaded and multithreaded mode.

10.6.1 Resuming and Suspending Threads

The only way software can suspend a thread is to execute an **mtspr** in hypervisor state that clears its corresponding Thread Enable bit to zero in the `CTRL` register (`CTRL[TE0]` or `CTRL[TE1]`).

Software executing in hypervisor state can resume execution for a suspended thread by setting the appropriate Thread Enable bit in the `CTRL` register (`CTRL[TE0]` or `CTRL[TE1]`). Hardware can resume a suspended thread with certain enabled interrupts. In either case, when the thread resumes, it begins execution at the System Reset interrupt vector location. See *Section 10.7.2.9* on page 320 for a detailed discussion of resuming and suspending threads.

10.6.2 Setting the Instruction-Dispatch Policy: Thread Priority and Temporary Stalling

Software running in a thread can influence how hardware dispatches the thread's instructions into the pipeline in two ways:

- The thread can set its priority level to low, medium, or high (depending on privilege state).
- The thread can execute a special **nop** instruction that suspends instruction dispatch for a specific number of processor cycles.

The priority level of one thread relative to the priority level of the other thread determines how the processor will allocate instruction dispatch bandwidth to the thread. If the thread has a higher priority than then the other thread, the hardware will, in general, dispatch the thread's instructions more frequently than it will dispatch the other thread's instructions. See *Section 10.5* on page 309 for a complete discussion of thread priorities and instruction-dispatch policy. The next section explains how a thread can change its priority and what changes are allowed.

A thread can also deliberately stall the dispatch of its instructions by executing a special **nop** instruction, as explained below.

10.6.2.1 *Allowed Thread Priority Changes*

Table 10-6 shows the changes to thread priority that a thread is allowed to make. The priority changes that are allowed depend on the privilege state of the thread and the priority-change bits in the `TSCR`. A thread must be in hypervisor or supervisor state to set its priority to high.

Table 10-6. Privilege States and Allowed Thread-Priority Changes

MSR[HV]	MSR[PR]	Privilege State	TSCR[PSCTP]	TSCR[UCP]	Can Suspend?	Thread Priority Change Allowed		
						High	Medium	Low
1	0	Hypervisor	Don't care	Don't care	Yes	Yes	Yes	Yes
0	0	Supervisor	0	0	No	—	—	—
			0	1	No	—	Yes	Yes
			1	Don't care	No	Yes	Yes	Yes
Don't care	1	User (Problem State)	Don't care	0	No	—	—	—
				1	No	—	Yes	Yes

A thread can attempt to change its priority level in two ways:

- The thread can write to its TSRL[TP] field with the desired priority encoding. If the change is not allowed, no action is taken. Writing '0' to this field will not disable the thread regardless of operating mode; writing '0' is always ignored.
- The thread can execute one of the special **nop** instructions listed in *Table 10-7* below that change thread priority as a side-effect.

By default, the hardware initializes the priority-change bits to allow priority changes only in hypervisor state. Hypervisor software can set the priority-change bits to allow software in other privilege states to change thread priority.

One thread cannot change the other thread's priority level. If software attempts this or some other priority change that is not allowed, the attempt is silently ignored by the processor.

If the TSCR[PBUMP] bit is set to '1' when an enabled interrupt is pending and not masked as a result of a system-caused exception, hardware will temporarily boost the thread's priority to medium if it was less than medium.

10.6.2.2 *nop Instructions that Change Thread Priority and Dispatch Policy*

In addition to the instruction-dispatch policy set by the combination of thread priorities, a thread can execute a special **nop** instruction to cause the processor hardware to suspend the dispatch of the thread's instructions for a specific number of cycles. Also, a thread can attempt to change its priority by executing one of a set of special **nop** instruction. See *Table 10-7*.

*Table 10-7. **nop** Instructions to Set Thread Priority and Stall Instruction Dispatch (Page 1 of 2)*

OR Form	Extended Mnemonic	Description/Side-Effects
`ori 0, 0, 0`	**nop**	Preferred form for **nop** instruction without side-effects.
`or 1, 1, 1`	`cctpl`	Change current thread priority to low.
`or 2, 2, 2`	`cctpm`	Change current thread priority to medium.
`or 3, 3, 3`	`cctph`	Change current thread priority to high.
`or 28, 28, 28`	`db8cyc`	Cause the current thread to block at dispatch for eight cycles. Used for multithread compiler optimizations of long-latency Vector/Scalar Unit (VSU) dependencies.

*Table 10-7. **nop** Instructions to Set Thread Priority and Stall Instruction Dispatch (Page 2 of 2)*

OR Form	Extended Mnemonic	Description/Side-Effects
`or 29, 29, 29`	`db10cyc`	Cause the current thread to block at dispatch for 10 cycles. Used for multithread compiler optimizations of long-latency VSU dependencies.
`or 30, 30, 30`	`db12cyc`	Cause the current thread to block at dispatch for 12 cycles. Used for multithread compiler optimizations of long-latency VSU dependencies.
`or 31, 31, 31`	`db16cyc`	Cause the current thread to block at dispatch for 16 cycles. Used for multithread compiler optimizations of long-latency VSU dependencies.

`ori 1-31, 1-31, 0`
`or 0, 0, 0`
`or 4-27, 4-27, 4-27`
	N/A	Non-special forms for **nop** instruction.

10.6.3 Preventing Starvation: Forward-Progress Monitoring

When one thread is assigned high priority and the other low priority, the low-priority thread will be given at most one chance out of `TSCR[DISP_CNT]` cycles to dispatch an instruction, and on that cycle, the low-priority thread will be denied the dispatch opportunity if the high-priority thread can use the dispatch slot. This situation can lead to complete starvation of the low-priority thread; it may make no forward progress for an excessively long time.

To prevent undesirable starvation, the PPE can be programmed to monitor the forward progress of each thread and take action when one thread has made no forward progress and the other has made more than a pre-set limit. The monitoring hardware uses the `TTR[TTIM]` field, the `TSRL[FWDP]` field, and the `TSCR[FPCF]` bit. Each thread has a dedicated set of these resources.

To start the hardware forward-progress monitor for a thread, an instruction sequence like the following can be used:

```
li       r10,0x1000    // other thread allowed 4096 instruction completions
mtspr    TTR,r10       // Set TTR[TTIM]
li       r10,(1<<19)   // FPCF bit position
mfspr    r11,TSCR      // get current TSCR
or       r11,r11,r10   // set the FPCF bit position to 1
mtspf    TSCR,r11      // set FPCF bit to 1 in TSCR
```

The forward-progress monitoring hardware will copy the `TTR[TTIM]` field into the `TSRL[FWDP]` field each time the thread successfully completes an instruction; thus, each time a thread completes an instruction, its corresponding forward-progress timer is reset to the initial value. In the above instruction sequence, when the second **li** instruction completes, the value in the `TTR[TTIM]` field—4096—will be copied into the `TSRL[FWDP]` field.

On cycles when the thread does not complete an instruction, the hardware will decrement the `TSRL[FWDP]` field by one if the other thread does complete an instruction; if neither thread completes an instruction, the `TSRL[FWDP]` value does not change. Thus, a thread's `TSRL[FWDP]` will be exhausted only if that thread completes no instructions while the other thread completes enough instructions to reduce the value in the `TTR[TTIM]` field to one.

The action taken when the TSRL[FWDP] field reaches one is determined by the TSCR[FPCF] bit: if this bit is set to '1', the other thread is flushed after the next instruction completes; if this bit is cleared to '0', no action is taken regardless of the value in TSRL[FWDP]. After the other thread is flushed, its dispatch is stalled until the starved thread completes an instruction.

The TSCR[FPCF] is only an enable for the hardware flush of the opposite thread; even when TSCR[FPCF] is cleared to '0' (flush of opposite thread disabled), the thread's TSRL[FWDP] field will decrement and be reloaded as described above (assuming the thread is enabled).

10.6.4 Multithreading Operating-State Switch

When software wishes to change the operating mode of the PPE from single-threaded to multi-threaded, or vice versa, a set of events must occur in the proper sequence to guarantee correct hardware and software behavior.

10.6.4.1 *Changing from Single-Threaded to Multithreaded Operation*

The steps, in hypervisor mode, to change the PPE operating mode from single-threaded to multi-threaded are as follows:

1. Perform a normal context-serializing operation such as **isync** that waits for all pipelines and queues to drain.
2. Issue the **mtspr** instruction for the CTRL register to enable the second thread.

10.6.4.2 *Changing from Multithreaded to Single-Threaded Operation*

The steps, in hypervisor mode, to change the PPE operating mode from multithreaded to single-threaded are as follows:

1. Perform a normal context-serializing operation such as **isync** that waits for all pipelines and queues to drain.
2. Disable interrupts and trace mode (MSR[EE], MSR[SE], and MSR[BE] cleared to '0').
3. Issue an **mtspr** CTRL instruction to disable your thread.

10.7 Pipeline Events and Instruction Dispatch

10.7.1 Instruction-Dispatch Rules

In a single-threaded processor implementation, instruction dispatch is governed by rules that consider many conditions in the processor such as:

- The availability of instructions to dispatch.
- The availability of execution units to process instructions ready for dispatch.
- Dependencies on previously dispatched instructions.
- Branch misprediction.

In the multithreaded PPE, instruction dispatch is governed by additional rules that consider thread-related conditions such as:

- Relative thread priority.
- A thread has made insufficient forward progress (forward-progress monitor decrements to a count of one).
- A thread experiences a cache miss.

The following sections discuss internal conditions that stall instruction dispatch.

10.7.2 Pipeline Events that Stall Instruction Dispatch

10.7.2.1 *Dependency on a Load Which Misses the DCache*

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

10.7.2.2 *Dependency on a Store-Conditional Instruction*

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

10.7.2.3 *Load or Store Instruction D-ERAT Miss*

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

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

10.7.2.4 *Microcoded Instruction*

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

10.7.2.5 *System-Caused Interrupt Pending*

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

10.7.2.6 *Forward-Progress Timer Timeout*

The forward-progress monitoring facility is described in *Section 10.6.3* on page 317.

When the `TSRL[FWDP]` count becomes one, the monitoring logic records that a forward-progress timeout condition has occurred. The `TSRL[FWDP]` counter continues to decrement while committed instructions of the opposite thread complete; the `TSRL[FWDP]` counter stops decrementing when it reaches x‘00001’. When the already committed instructions complete, the opposite thread will be flushed and dispatch will be blocked until the starved thread completes an instruction.

10.7.2.7 *Caching-Inhibited Load Issued with Precise Machine Check Interrupts Enabled*

If precise Machine Check interrupts are enabled (`HID0[en_prec_mchk] = 1`) and a caching-inhibited (`I = 1`) load instruction is issued, then all younger instructions for the thread that issued the load are flushed, refetched, and then stalled at dispatch. The stall is released when the caching-inhibited load instruction completes.

10.7.2.8 *Thermal Throttling*

When the thermal sensors indicate the chip is overheating, or in order to save power, the processor may detect and signal a thermal throttling condition. When signalled, dispatch is blocked for one or both threads.

10.7.2.9 *FPU Busy*

When the FPU is executing a long-latency floating point instruction (**fdiv** or **fsqrt**) or any floating point instruction whose operand is denormalized number, the PPE recycles the operation in the FPU issue queue, which causes a stall that may propagate back to dispatch.

10.7.2.10 *Sync*

A **sync** (`L = 0` or `1`) instruction blocks both threads until the **sync** instruction completes. The **eieio** and **tlbsync** instructions do not cause threads to stall. The **isync** instruction stalls both threads until all previous instructions complete, then the **isync** causes a flush and refetch on it's own thread. The instructions from the other thread that are already fetched or dispatched are not refetched after the **isync** completes.

10.8 Suspending and Resuming Threads

By adjusting thread priorities and other instruction-dispatch controls, software can reduce the rate of forward progress of a thread so that it is active but nearly disabled. To disable a thread completely, software can clear to '0' the thread's `CTRL[TE0]` or `CTRL[TE1]`. When a thread is disabled, the fetch hardware will not fetch instructions into its instruction buffer, and the instruction-dispatch logic will not consider it for priority when making decisions for instruction dispatch. This means that the other, enabled thread (if not in low priority) will be granted all available instruction-fetch and instruction-dispatch bandwidth.

10.8.1 Suspending a Thread

The only way software can suspend a thread is to execute the **mtctrl** instruction with an operand that leaves the thread's Thread Enable bit cleared to '0'.

A write to `CTRL[TE0]` or `CTRL[TE1]` will succeed only when the thread is executing in hypervisor state (`MSR[PR,HV] == 0,1`). If the write is attempted in supervisor or problem state, the write is ignored silently. Attempting to clear the Thread Enable bit for the thread that is not executing the **mtctrl** instruction will be ignored, regardless of the thread's privilege state.

If a thread suspends itself and the other thread remains active, the thread suspension will be treated as a context-synchronizing event for the active thread.

10.8.2 Resuming a Thread

The actions taken to resume a suspended thread are essentially the same as the actions taken to start the initial thread after the Power-On-Reset (POR) sequence has completed. In both cases, the thread resumes execution at the System Reset interrupt vector location. The System Reset interrupt handler can determine whether the System Reset handler is running because of a POR event or a normal thread-resuming event and handle the event appropriately. The handler makes the determination based on the values in the Save and Restore Registers.

10.8.2.1 *Thread-Resuming Events*

A previously suspended thread can be resumed by any of the following events:

- A thread executing in hypervisor state (`MSR[PR,HV] == 0,1`) can resume the other thread by setting to one the corresponding Thread-Enable bit (`CTRL[TE0]` or `CTRL[TE1]`) with the `mtctrl` instruction.
- The occurrence of a Thermal Management interrupt can resume a thread if the `HID0[therm_wakeup]` bit is set to '1'.
- The occurrence of a System Error interrupt can resume a thread if the `HID0[syserr_wakeup]` bit is set to '1'.
- The occurrence of an External interrupt can resume a thread if the `TSCR[WEXT]` bit is set to '1'.
- The occurrence of a Decrementer interrupt for the suspended thread can resume the thread if the corresponding Decrementer Wakeup Enable bit (`TSCR[WDEC0]` or `TSCR[WDEC1]`)is set to '1'.

A Machine-Check interrupt cannot occur for a suspended thread, and while a Hypervisor Decrementer interrupt can occur for a thread, it does not resume a suspended thread.

When a system-caused exception occurs, the thread resumes even if the interrupt that corresponds to the exception is masked.

After POR, the following interrupt mask bits are set to mask interrupts:

- `HID0[therm_wakeup,syserr_wakeup] = '00'`.
- `TSCR[WDEC0,WDEC1] = '00'`.
- `TSCR[WEXT] = '0'`.

With these mask bits cleared to '0', a suspended thread will not resume on any of the corresponding exceptions.

10.8.2.2 *Hardware Resume Process*

A resume(d thread starts execution at the System Reset interrupt vector location. Hardware sets the `TSRL[TP]` field to `'11'`, which sets the thread priority to high. The thread priority is set to high regardless of the condition that caused the thread to resume.

When the resumed thread begins executing at the System Reset interrupt vector location, the System Reset Reason Code field of the Machine Status Save Restore Register 1 (`SRR1[42:44]`) is set to one of the thread resume reason codes shown in *Table 10-8*.

Table 10-8. Thread Resume Reason Code

`SRR1[42:44]`	Resume Reason	Condition
101	System Reset	Write to `CTRL[TE]` field
010	Thermal Management	`HID0[therm_wakeup]`
110	System Error	`HID0[syserr_wakeup]`
100	External	`TSCR[WEXT]`
011	Decrementer	`TSCR[WDEC]`

If multiple reasons exist for the thread to resume, the reason code reported in `SRR1[42:44]` will respond to one of the reasons that caused the wakeup (there is no predetermined priority).

The fields in `SRR1` other than the System Reset Reason Code contain values from the `MSR` as if the thread had never been suspended. The Machine Status Save/Restore Register 0 (`SRR0`) similarly contains the address of the instruction that would have been executed next if the thread had not been suspended.

The System Reset interrupt handler can use the information in `SRR0` and `SRR1` to resume the suspended thread properly. Since the same reason code is used for both System Reset resume and a software-initiated `mtctrl` resume, the System Reset interrupt handler must examine the value of `HID0` or `HID1` to determine the cause of the thread resume action.

Since a System Reset event can happen only at POR, if software finds `HID0` or `HID1` set to its default POR values, software knows the thread resume was the result of a System Reset event. If software finds that the values in the `HID` registers differ from their POR default values, software knows the thread resume was the result of a **mtctrl** instruction.

10.8.2.3 *Accepting Interrupts after Resuming a Thread*

When a thread is resumed by a Thermal Management, System Error, External, or Decrementer interrupt, the resuming interrupt is then masked because `MSR[EE,HV]` will be set to ‘01’ when the resumed thread is started at the System Reset interrupt vector location. If software later enables the interrupt by changing the `MSR` settings and the exception is still pending, the interrupt that caused the thread to be resumed will be taken. In other words, a resuming interrupt will cause a suspended thread to resume but will not cause the resumed thread to take the resuming interrupt until the resumed thread enabled interrupts.

When two threads are resumed in the same clock cycle (a rare event), it is indeterminate which thread is resumed first, but both threads will resume properly. When one thread is active and the other is inactive and the inactive thread is resumed, the active thread will view the thread resumption as a context-synchronizing event.

Between the time the thread is resumed and the time the thread enables interrupts, additional exceptions can occur. The interrupt that will be taken after the thread enables interrupts is not necessarily the interrupt that caused the thread to resume; the interrupt taken by the thread will be determined according to the interrupts pending and the interrupt priorities at the time the interrupt is taken.

10.8.3 Exception and Interrupt Interactions With a Suspended Thread

External, Precise Machine Check, System Reset, Thermal Management, and System Error interrupts are thread dependent and handled by the thread that causes the exception. These interrupts are enabled for both threads, respectively, by `TSCR[WEXT]`, `HID0[en_prec_mchk]`, `HID0[en_syserr]`, `HID0[therm_wakeup]`, and `HID0[syserr_wakeup]`.

10.8.3.1 *Decrementer Interrupt*

Each thread has a separate Decrementer register that creates a separate Decrementer interrupt for the corresponding thread. `TSCR[WDEC0]` or `TSCR[WDEC1]` enables waking up a suspended thread due to a Decrementer Interrupt.

10.8.3.2 *Hypervisor Decrementer Interrupt*

The single Hypervisor Decrementer Register is shared by both threads. A Hypervisor Decrementer interrupt does not resume a suspended thread.

10.8.3.3 *Imprecise Machine Check Interrupt*

Imprecise Machine Check interrupts do not have thread information associated with them. All enabled threads take an imprecise Machine Check interrupt regardless of which thread caused the exception. The `MSR[ME]` bit must be set to ‘1’ for both threads (even suspended threads) to avoid causing a checkstop. An imprecise Machine Check interrupt does not resume a suspended thread. If the thread is later resumed, it will not take an imprecise Machine Check interrupt.

10.8.3.4 *Maintenance Interrupt*

An Instruction-Caused Maintenance interrupt is thread-dependent and is handled by the thread that caused the exception. A System-Caused Maintenance interrupt is shared by both threads and occurs when one or both of the two trace input signals, which are shared by both threads, is asserted. The System-Caused Maintenance interrupt is handled by the thread that enables the interrupt.

10.8.3.5 *Floating-Point Enabled Exceptions*

When either floating-point exception mode bit (`MSR[FE0]` and `MSR[FE1]`) is set for either thread (it does not matter which thread is executing), the PPU switches into a single-instruction-issue mode. In this mode, only one instruction is active in execute at a time, and all other instruction from either thread are held at issue until the previous instruction completes.

10.8.3.6 *Other Interrupts*

All other interrupt are thread-dependent and are handled by the thread that caused the exception. For details, see *Section 9 PPE Interrupts* on page 237.

10.8.4 Thread Targets and Behavior for Interrupts

When an exception is signalled or an interrupt is taken, the PPE interrupt logic selects a thread to handle the event. The chosen thread depends on the exception type and the interrupt type if the exception results in an interrupt. *Table 9-38* on page 287 shows the target thread(s) for each exception and interrupt type.

The thread that the interrupt logic selects as the target can be in the suspended state or enabled state. If the thread is enabled, it will take the interrupt normally. If the thread is suspended, it will either be resumed or allowed to remain suspended.

Table 10-9 shows the entry condition, activated thread, mask condition, and suspended-thread behavior for each interrupt type. For example, a machine-check interrupt will target both threads, but it will be recognized only by the threads that are enabled; suspended threads are not resumed by machine-check interrupts. In this table, a "—" symbol means "not applicable".

Table 10-9. Interrupt Entry, Masking, and Multithreading Behavior (Page 1 of 2)

Interrupt Type	Entry Condition	Activated Thread	Mask Condition	Suspended Thread Behavior
System Reset	POR	thread 0	—	If SRR1[42:44] = '101', thread resumed due to system reset.
	wakeup	specific thread	—	This exception is where suspended thread resumes by means of a write to `CTRL`[TE01] and SRR1[42:44] = '101'
Machine Check	MC interrupt	all active threads	MSR[ME]	Remain suspended.
Data Storage Interrupt	memory access	specific thread	—	—
Data Segment Interrupt	memory access	specific thread	—	—

Table 10-9. Interrupt Entry, Masking, and Multithreading Behavior (Page 2 of 2)

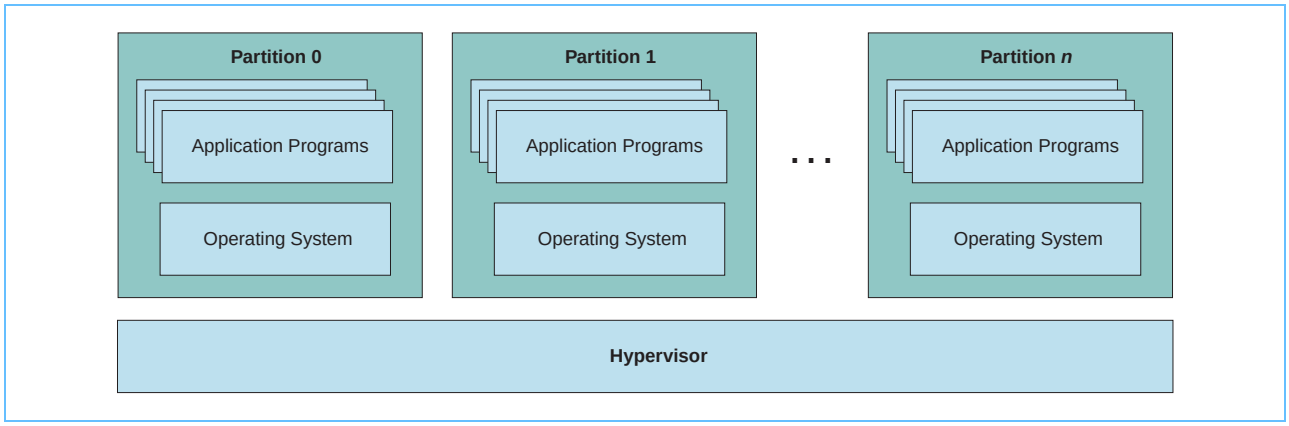
Interrupt Type	Entry Condition	Activated Thread	Mask Condition	Suspended Thread Behavior
Instruction Storage Interrupt	memory access	specific thread	—	—
Instruction Segment Interrupt	memory access	specific thread	—	—
External Direct	external interrupt if HID0[extr_hsrr] = ’0’	programmable	MSR[EE]	If `TSCR[WEXT]` = ’1’, wake up at System Reset and set SRR1[42:44] = ‘100’.
External Direct (Mediated mode)	external interrupt if HID0[extr_hsrr] = ’1’	programmable	MSR[EE] | (LPCR[LPES] bit 0 = ’0’ & MSR[HV] = ’0’) | MSR[SV]	If `TSCR[WEXT]` = ’1’, wake up at System Reset and set SRR1[42:44] = ‘100’.
External Mediated	HID0[extr_hsrr] = ’1’ & LPCR[MER] = ’1’	programmable	MSR[EE] & (!MSR[HV] | MSR[PR])	If `TSCR[WEXT]` = ’1’, wake up at System Reset and set SRR1[42:44] = ‘100’.
Alignment	misaligned access	specific thread	—	—
Program	Invalid operation	specific thread	—	—
Floating-Point Unavailable	any floating operation if MSR[FP] = ’0’	specific thread	—	—
Decrementer	decrementer timeout	specific thread	MSR[EE]	If TSCR[WDEC] = ’1’, wake up at System Reset and set SRR1[42:44] = ‘011’.
Hypervisor Decrementer	hypervisor decrementer timeout	all active threads	MSR[EE] | !MSR[HV] & LPCR[HDICE]	remain suspended
System Call	**sc** instruction	specific thread	—	—
Trace	any operation if MSR[SE] = ’1’ or MSR[BE] = ’1’	specific thread	—	—
VXU Unavailable	VXU operation if MSR[VXU] = ’0’	specific thread	—	—
System Error	Machine Check outside PPE if HID0 [en_ mchk] = ‘1’	all active threads	MSR[EE]& !MSR[HV]	If `HID0[syserr_wakeup]` = ’1’, wake up at System Reset and set SRR1[42:44] = ‘110’.
Maintenance	instruction caused (HSRR1[33] = ’0’)	specific thread	—	—
Maintenance	system (trace sig) (HSRR1[33] = ’1’)	all threads	MSR[EE]& !MSR[HV]	—
Thermal Management	thermal condition	all active threads	HID0[therm_intr_en]	If `HID0[therm_wakeup]` = ’1’, wake up at System Reset and set SRR1[42:44] = ‘010’.

11. Logical Partitions and a Hypervisor

11.1 Introduction

Logical partitioning refers to the capability to logically subdivide a single system into multiple partitions, each capable of running its own operating system and user environment such that a program executing in one partition cannot interfere with any program executing in a different partition. *Figure 11-1* illustrates the concept of logical partitioning and the hypervisor partition-management software that controls it.

Figure 11-1. Overview of Logical Partitions



Logical partitioning can be useful for several purposes, including:

- In software-production systems, to temporarily dedicate system resources to a testing effort for new software or applications before bringing them fully online in mission-critical environments.
- In business environments to reduce cost by means of system consolidation; the consolidation involves moving workloads from many systems to a single system which is logically partitioned to mimic the environment of the many systems.
- In consumer-electronic and embedded systems, to support environments such as a real-time operating system executing in one partition, a conventional operating system executing in another partition, and a security subsystem executing in a third partition.
- In aggregated computer centers, Internet service providers, and application-service providers, where fractional provisioning of computing and other system resources can support many different and perhaps competing workloads.
- In mission-critical applications, to support a complete hot-plug architecture, including hot-plug replacement of DRAM modules and I/O devices without the operating system being involved or even knowing what happened.

In these environments, separation of applications and data must be maintained while the required performance and system resources are provided for each partition. Reduced cost through logical partitioning can be realized, because hardware resources can be supplied to

each partition, and dynamic provisioning can be used to efficiently move resources from lightly-used partitions to those that are experiencing high or peak demand. This kind of resource allocation reduces a system's requirement to over-provision in order to meet all worst-case loads.

11.1.1 The Hypervisor and the Operating Systems

In a logically-partitioned system, partition-management software—called a *hypervisor*—creates the partitions by assigning portions of the total system resources (processor-element execution time, real memory, and I/O devices) to each partition. The hypervisor executes at a more-privileged level than an operating system in a partition.

Separate instances of an operating system, or different operating systems, can execute in each partition. The hypervisor, with hardware assistance, ensures that the operating system and applications in one partition have no knowledge of, or access to, the resources of another partition, unless specifically granted by the hypervisor. Exceptions to this rule can be made in which the hypervisor brokers a communication between partitions, or one partition publishes services that can be used by other partitions.

11.1.2 Partitioning Resources

System resources are typically statically assigned by a primary partition to other partitions, and/or by an administrator's use of a hardware management console (typically a serial port). In these environments, the policy of how many partitions to create, and with what resources, is separated from the mechanism for partitioning. For consumer-electronic or embedded-system applications, for example, a primary partition can first be created by the hypervisor. This partition can then use hypervisor interfaces to create and allocate resources for other partitions. Dynamic-provisioning techniques can move resources from one partition to another, based on demand and the need to meet user-level requirements, such as interactive or real-time response guarantees.

The hypervisor is typically implemented in a small executive that is packaged as firmware. It provides a newly-created partition with access to resources and it harvests those resources when that partition is deleted.

If the hypervisor is unable to dedicate an entire physical resource to a partition, it can virtualize the physical resource into a set of logical resources that are time-shared. For example, if a fraction of the PPE is provided to a partition, the hypervisor can manifest the physical PPE as multiple virtual PPEs that are assigned to the logical partitions in time-multiplexed fashion—for example, by using Hypervisor Decrementor (HDEC) interrupts to drive the transitions. If an I/O device cannot be logically partitioned, such that each operating system has direct access to its assigned I/O, the hypervisor can virtualize the I/O device so that the device can be shared, or so that an I/O-hosting partition can own the physical device and host other logical partitions' I/O requirements.

The CBE processor provides hardware facilities for logical partitioning in the PPE, the SPEs, the I/O controller, and the Element Interconnect Bus (EIB). The standard PowerPC-based hypervisor logical-partitioning facilities are supported, as well as extended facilities for the SPEs, I/O resources, and EIB. The CBE processor also supports a Resource Allocation Management (RAM) facility that can be used in conjunction with the logical-partitioning facilities to control the amount of bandwidth allocated to units in partitions accessing shared resources. The RAM facility is described in *Section 8 Resource Allocation Management* on page 203.

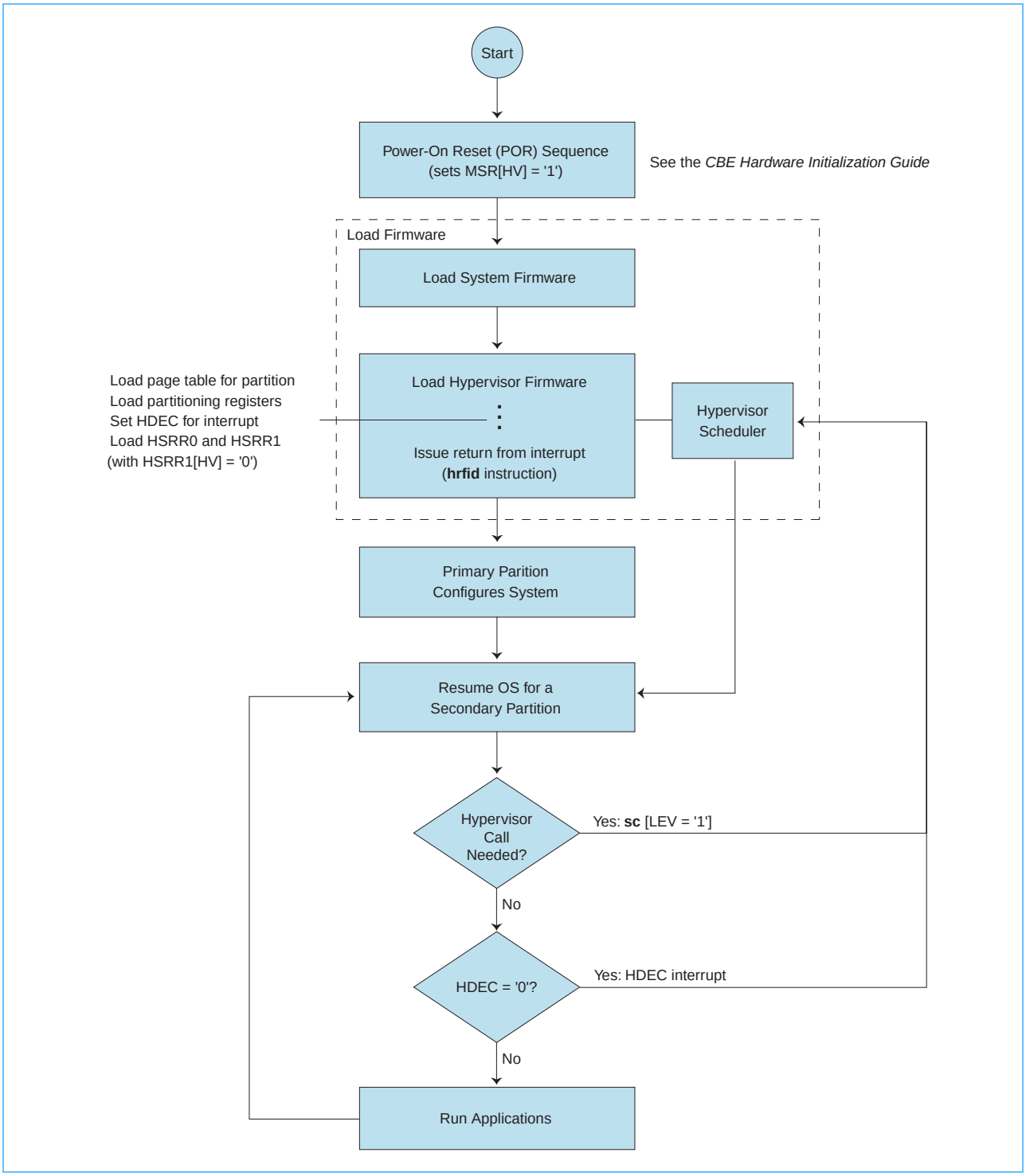
11.1.3 An Example Flowchart

Figure 11-2 on page 330 shows how a hypervisor might boot and manage logical partitions. The events shown here cover only the most basic workings of a hypervisor, the primary partition (also called an alpha or I/O-hosting partition), and a secondary partition and its applications. Other methods of implementing a hypervisor are possible.

The System Reset interrupt can be used for Power-On Reset (POR). This interrupt sets both the Hypervisor bit, `MSR[HV]`, and the 64-bit mode bit, `MSR[SF]`, to '1', so that the CBE processor comes out of POR in hypervisor mode. After loading system firmware, the CBE processor loads the hypervisor firmware. This in turn loads the hardware-managed page table for the first partition (the hypervisor maintains such a page table for each partition), the registers used for hypervisor control of the partition, the HDEC decrementer, and the `HSSR0` and `HSSR1` registers that contain the program counter and `MSR` values for the first partition. The hypervisor then executes a return from interrupt (**hrfid**) to begin execution of the first partition.

The primary partition is the first partition created by the hypervisor. This partition has a small operating system (OS) and its mission is to discover the resources in the system (memory, I/O, and so forth), and manage the creation of secondary partitions for the hypervisor. After the primary partition has configured the system, it passes control to the OS of a secondary partition. This partition's OS runs its applications until it either needs help from the hypervisor (such as an address translation) or until its time-share slice is finished (signaled by the HDEC count of '0'). It obtains help from the hypervisor by executing a *hypervisor call* (hcall), which is the **sc** instruction with the LEV bit = '1'. When either of these events occur, the hypervisor loads the necessary information (such as address-translation information) and returns control to the partition, again by executing a return from interrupt.

Figure 11-2. Flowchart of overview of Hypervisor Management of Logical Partitions



11.2 PPE Logical-Partitioning Facilities

The PPE's standard PowerPC-based logical-partitioning architecture is defined in the *PowerPC Operating Environment Architecture, Book III*. The PPE can be assigned to one partition at a given time. If active, both PPE threads execute in the same logical partition.

11.2.1 Enabling Hypervisor State

The *PowerPC Architecture* supports three privilege states, which are controlled by `MSR[HV]` and `MSR[PR]` bits. MMIO registers are classified according to these states. The states are designed for the following uses:

- *Hypervisor State*—`MSR[HV]` = '1' and `MSR[PR]` = '0'. The most-trusted state, used by the hypervisor. Access to all system facilities is provided at this level of privilege. In the *PowerPC Architecture*, this state is occassionally referred to as the "privileged and hypervisor state"; for convenience, this document always refers to the state as "hypervisor state".
- *Privileged State*—`MSR[HV]` = '0' and `MSR[PR]` = '0'. Used by the operating system within a logical partition. In the *PowerPC Architecture*, this state is occassionally referred to as the "privileged and non-hypervisor state" or "supervisor state"; for convenience, this document always refers to the state as "privileged state".
- *Problem State*—`MSR[HV]` = '0' and `MSR[PR]` = '1', or `MSR[HV]` = '1' and `MSR[PR]` = '1'. The least-trusted state, used by application software in a logical partition.

The `MSR[PR]` bit can be set using the **mtmsr** instruction. Privileged or hypervisor state is entered by executing the **sc** (system call) instruction or as a result of some interrupts. The **sc** instruction has a LEV bit which, when set to '1', causes `MSR[HV]` to be set to '1'. Exiting the privileged or hypervisor states is accomplished by executing the **rfid** or **hrfid** instruction. For details, see *Table 9-40* on page 290 and the *PowerPC User Instruction Set Architecture, Book I* and *PowerPC Operating Environment Architecture, Book III*.

If no hypervisor exists on the CBE processor, the entire system is normally run at `MSR[HV]` = '1', and only two privilege states exist, `PR` = '0' for firmware and operating-system privileged state, and `PR` = '1' for application-software problem state.

If address-translation is enabled, privileged software can control, by means of page-table entries, whether application software is given access to particular problem-state MMIO registers; access to the MMIO registers in this mode is not directly enforced by hardware.

11.2.2 Hypervisor-State Registers

The following registers are used by the hypervisor to configure operations in logical partitions. Most of these registers can only be accessed in hypervisor state:

- *Logical-Partition Identification Register (`LPIDR`)*—Specifies the logical partition to which the PPE (both threads) is currently assigned. Up to 32 partitions may be used.
- *Processor Identification Register (`PIR`)*—Specifies the system-wide processor ID for each PPE thread.
- *Hypervisor Status Save and Restore Registers (`HSRR0, HSRR1`)*—Specifies the state of the PPE when an interrupt occurs, and/or the state from which the PPE should resume after a hypervisor return from interrupt (**hrfid**).

- *Storage Description Register 1 (SDR1)*—Specifies the real address and size of the hardware-accessed page table for the current partition. The hypervisor provides each partition with its own hardware page table.
- *Real-Mode Offset Register (RMOR)*—Specifies the offset from real address x'0' at which real-mode memory begins for non-hypervisor real-mode accesses. See *Figure 11-3* on page 334 for how an OS and the hypervisor view this memory space.
- *Logical-Partitioning Control Register (LPCR)*—Specifies the largest effective address that can be used by a logical partition in real mode, how storage is accessed and cached in real mode, and certain effects of interrupts. See *Figure 11-3* on page 334 for how an OS and the hypervisor view the real-mode address limit.
- *Hypervisor Real-Mode Offset Register (HRMOR)*—Specifies the offset from real address x'0' at which real-mode memory begins for hypervisor real-mode accesses.
- *Hypervisor Decrementer (HDEC)*—Provides a means for the hypervisor to manage timing functions independently of the Decrementer (DEC) registers, which are managed by user or supervisor software running in a partition. Like the DECs, the HDEC provides a means of signaling an interrupt after a specified amount of time has elapsed. This is useful, for example, when time-sharing resources.
- *Machine-Check Enable Bit (MSR[ME])*—Enables Machine Check interrupts.
- *Time Base (TB) Register*—Specifies the value of the time base.
- *Data-Address Breakpoint Register and Data-Address Breakpoint Extension Register (DABR, DABRX)*—Specifies the effective address and other conditions of a load or store access at which a Data Storage exception should occur.
- *Hypervisor Software-Use Special-Purpose Registers (HSPRG0, HSPRG1)*—Available for any use by the hypervisor.
- *Hardware-Implementation-Dependent (HID) Registers*—Specify a variety of PPE-state and debug functions.

See the *Cell Broadband Engine Registers* specification for details about these registers. See the *PowerPC Operating Environment Architecture, Book III* and *Section 20 Shared-Storage Synchronization* on page 551 for the required synchronization instructions and sequences that must be used by the hypervisor with the above facilities. Failure to properly synchronize these changes can lead to inconsistent state and failure of the application or hypervisor.

11.2.3 Controlling Real Memory

When supporting logical partitioning, the hypervisor must itself use some real memory and also provide some real memory to each logical partition. The hypervisor must control access to real memory so that no logical partition may access another partition's real memory or the hypervisor's real memory.

In the Instruction Relocate mode (MSR[IR]) and Data Relocate mode (MSR[DR])—described in the *Cell Broadband Engine Registers* specification and the *PowerPC Operating Environment Architecture, Book III*—this access to real memory is controlled by the hypervisor, which is responsible for the hardware-accessed page table and Translation Lookaside Buffer (TLB) for each partition. This hardware-accessed page table is described by the contents of the Storage Description Register 1 (SDR1). The SDR1, the TLB facilities, and the real memory reserved by the hypervisor for the hardware page table must be accessible only to the hypervisor.

The operating system in a logical partition normally manages the virtual-address (VA) space for the logical partition and the mapping between effective-address (EA) space and VA space through direct manipulation of its Segment Lookaside Buffer (SLB). However, the relationship between the VA space and the real-address (RA) space is managed by the operating system indirectly through hypervisor calls.

The hypervisor typically uses its own real-memory allocation table to manage which blocks of real memory have been allocated to a logical partition. As shown in *Figure 11-3* on page 334, the hypervisor allocates to a partition the real memory that the partition software uses as address 0, and this memory appears (to the partition software) contiguous up to the total amount of memory allocated to the partition.

When managing its own real memory, the operating system in a logical partition typically makes a hypervisor call to request a page-table update for its VA-to-RA mapping. The hypervisor then checks the requested real-page mapping, and if valid for that logical partition, it converts the RA supplied by the operating system to the true RA and manages the required changes to the hardware page table and TLBs.

The PPE and the SPE MMUs support three simultaneous pages sizes—4 KB, and two sizes selectable from 64 KB, 1 MB, or 16 MB. Large page sizes are very useful in reducing thrashing of TLBs when applications access large amounts of main storage. The PPE provides 1,024 TLB entries and each SPE provides 256 TLB entries.

11.2.3.1 ***Memory-Management Facilities***

The hypervisor typically executes in real mode (`MSR[IR]` = ‘0’ for instructions, `MSR[DR]` = ‘0’ for data), in which the real-mode (non-relocate) translation facility is employed. However, some hypervisor implementations may choose to have components that operate with instruction and/or data relocation enabled (`MSR[IR]` = ‘1’ and/or `MSR[DR]` = ‘1’). When relocation is to be enabled, the hypervisor can choose to have VA- to-RA translations supported by the hardware page table by setting `LPCR[TL]` = ’0’, or it can specify the software-managed TLB mode by setting (`LPCR[TL]` = ’1’). This can be varied on a logical-partition basis.

Software-Managed TLB Mode

In software-managed TLB mode, software directly manages the TLBs and the hardware MMU never directly accesses a page table in main storage in order to reload a TLB entry. An Instruction Storage or Data Storage interrupt is generated immediately following any TLB miss by the VA-to-RA translation hardware (see *Section 11.2.4 Controlling Interrupts and Environment* on page 337 and *Section 11.3.3 Controlling Interrupts* on page 343).

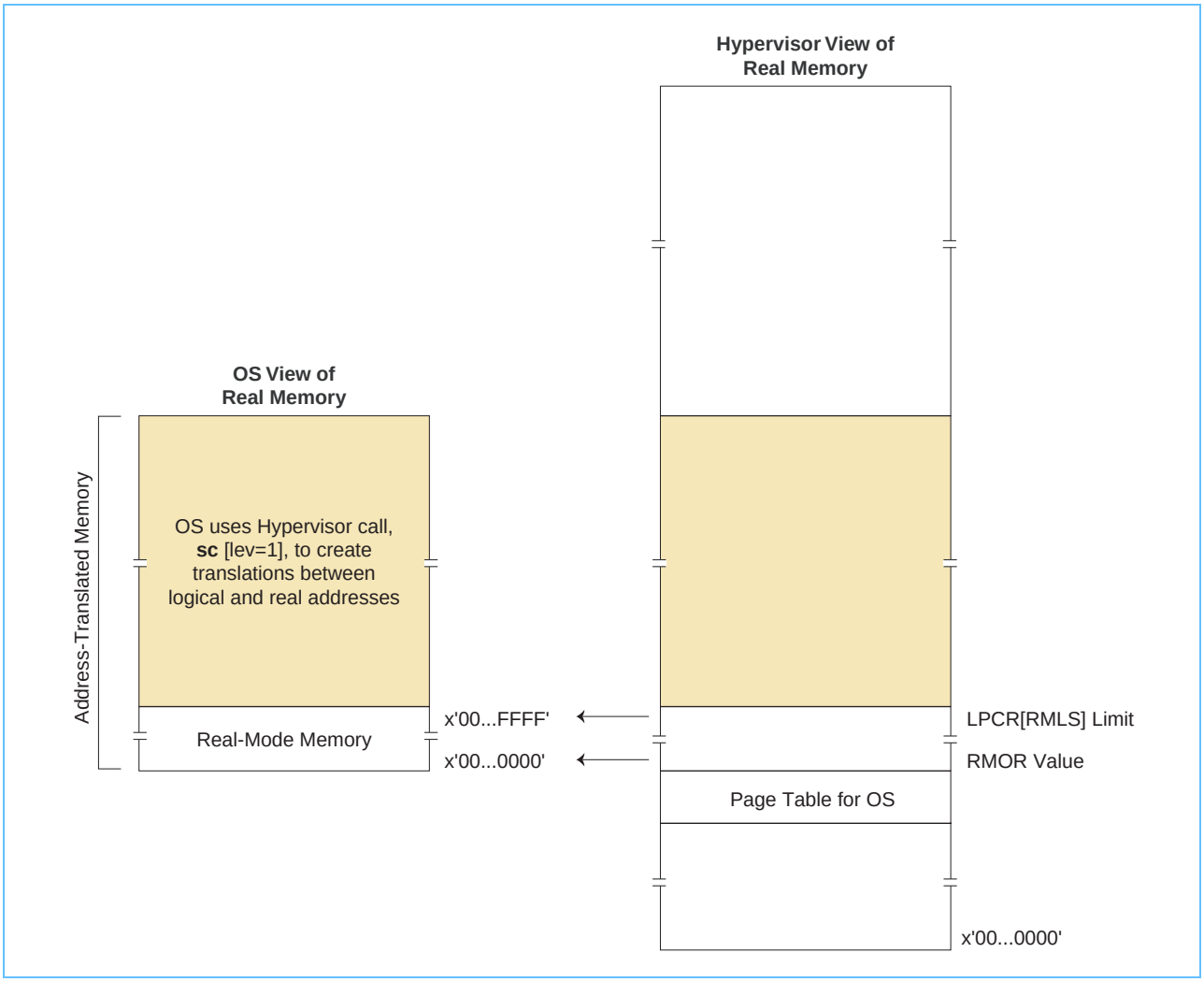
Hypervisor software uses several registers to directly manage the TLBs, including the PPE TLB Index Hint Register (`PPE_TLB_Index_Hint`), PPE TLB Index Register (`PPE_TLB_Index`), PPE TLB Virtual-Page Number Register (`PPE_TLB_VPN`), and PPE TLB Real Page Number Register (`PPE_TLB_RPN`). In addition to using these registers to reload a TLB entry after a miss, software is also able to preload translations directly into the TLB with these registers for improved real-time response.

The **tlbie(l)** instruction must be used to invalidate the contents of a TLB entry in both hardware- and software-managed TLB modes. The hypervisor also uses Hardware Implementation-Dependent Register 6 (`HID6`) to select the sizes of up to two concurrent large page sizes, if large-page translations are to be supported by the hypervisor and/or provided to one or more partitions.

Access by Operating Systems to Physical Memory in Real Mode

Many existing operating systems require some access to physical memory in real mode, typically an area in low memory for interrupt handling and typically starting at real address x'0'. Unrestricted access to physical memory by a logical partition operating in real mode cannot be allowed. This is handled through the use of the Real Addressing Mode (Real Mode) facility, which includes (among other resources) the Real Mode Offset Register (`RMOR`) and the Real Mode Limit Selector field in the Logical-Partitioning Control Register (`LPCR[RMLS]`) when `MSR[HV]` = '0' and `LPCR[LPES]` bit 1 = '1'. Using these facilities, the hypervisor can provide a logical partition with access to a limited amount of physical memory while accessing storage in real mode, as shown in *Figure 11-3* on page 334.

Figure 11-3. Operating-System and Hypervisor Views of Physical Memory



Access by Hypervisor to Physical Memory in Real Mode

A similar facility exists, using the Hypervisor Real Mode Offset Register (HRMOR), for the hypervisor itself to provide a real-mode offset when accessing storage in real mode with MSR[HV] = '1'. This can be useful in certain system configurations—such as multiprocessor Non-Uniform Memory Access (NUMA) environments—in which a copy of hypervisor code or data should be replicated for performance or reliability considerations. The HRMOR value is only effective when bit 0 (the most-significant bit) of the EA is '0'. If bit 0 of the effective address is a '1', bits 2 through 63 of the EA are used as the RA for the access.

Real-Mode Limit Select

The Real Mode Limit Selector field in the Logical-Partitioning Control Register, LPCR[RMLS], controls the size of the partition's real-mode offset. The Logical Partition Environment Selector field, LPCR[LPES] (see *Section 11.2.4.1* on page 337), controls how External Interrupts and a subset of internal interrupts are handled.

Table 11-1 summarizes the supported Real Mode Limit Select (RMLS) values and the corresponding limits chosen. The CBE processor supports an additional high-order bit of the RMLS field not listed in *PowerPC Operating Environment Architecture, Book III*, for a total of four bits rather than three. When in Real Offset Mode and the requested effective address is beyond the chosen RMLS limit, an Instruction or Data Storage interrupt occurs, and SRR1[36] is set to '1'.

Table 11-1. Summary of Real Mode Limit Select (RMLS) Values

LPCR[RMLS] Field	Effective Address Limit
0000	256 GB
0001	16 GB
0010	1 GB
0011	64 MB
0100	256 MB
0101	16 MB
0110	8 MB
0111	128 MB
1000	Reserved[1]
1001	4 MB
1010	2 MB
1011	1 MB
11xx	Reserved[1]

1. All reserved values are treated as 1 MB.

Real-Mode Caching

The Real Mode Caching-Inhibited bit in the `LPCR` register (`LPCR[RMI]`) controls whether a real-mode data access is treated as cacheable or caching-inhibited. When the Real Mode Storage Control facility (`HID6[RMSC]`) is used, the `LPCR[RMI]` bit is only used for real-mode accesses with addresses above the address specified in the `HID6[RMSC]` field. This control is used to avoid a cache paradox (the inadvertent caching of non-cacheable data) when accessing data storage in real mode.

Logical Partition Identification Register

The PPE has a Logical Partition Identification Register (`LPIDR`) that the hypervisor writes to assign the PPE to a partition. In a multi-CBE-processor system, or when the PPE is time-shared or virtualized so that it can serially support multiple partitions (fractional assignment of the PPE), the hypervisor updates the `LPIDR` register with the identifier of the logical partition to which the PPE has been assigned. This value is used by the hardware to avoid unnecessary removal of TLB entries.

The `LPIDR` register is used by the hardware under three circumstances:

- To tag TLBs with the current Logical Partition ID (LPID) value when the TLB entry for a page translation is loaded by the hardware.
- To invalidate only TLB entries that match the criteria of the **tlbie(l)** instruction when the current `LPIDR` contents match the LPID tag value of the matching entries in the TLB. This prevents a previous partition's TLB entries from being discarded from the TLB unnecessarily by the partition currently using the PPE. This also prevents a broadcast **tlbie** instruction issued on a processor element from unnecessarily invalidating TLB entries on another processor element that is operating in a different partition.
- To prevent the hardware from using a TLB entry for translation when the current contents of the `LPIDR` does not match the LPID tag of the TLB entry. This prevents a partition from using a different partition's address translation. It also removes the requirement for the hypervisor to invalidate all TLB entries when switching the PPE between partitions. However, the LPID tag in the TLB entry does not affect the hardware's TLB replacement algorithm.

11.2.3.2 ***Real-Time Memory Management***

In addition to using standard cache-management and TLB-management instructions, the hypervisor can also control the replacement policy for cache and TLB sets on behalf of itself or at the request of the operating system in a logical partition. Hardware uses a least-recently used (LRU) algorithm to determine which cache line or TLB entry to replace. In many real-time applications, such replacement can result in non-deterministic behavior or poor performance, due to application-specific access patterns. One solution is to use the software-managed TLB mode, described in *Section 11.2.3.1 Memory-Management Facilities* on page 333. Another is the software locking of TLB entries, as described in *Section 4.2.7.7 TLB Replacement Management Table* on page 98.

Data- and Instruction-Address Range Registers

The hypervisor has access to the PPE's Address Range Registers, which support the pseudo-LRU replacement algorithm for L2 and TLB replacement. An Address Range is a naturally-aligned range that is a power-of-2 size between 4 KB and 4 GB, inclusive.

Each hardware thread has two sets of Data-Address Range Start Registers (`DRSR0` and `DRSR1`), Data-Address Range Mask Registers (`DRMR0` and `DRMR1`), and associated Class ID registers. Each hardware thread also has two sets of Instruction Address Range Start Registers (`IRSR0` and `IRSR1`), Instruction Address Range Mask Registers (`IRMR0` and `IRMR1`), and associated Class ID registers.

These registers define four windows of EA space (two for data accesses and two for instruction fetches) in which the accesses can be associated with a replacement-class identifier. This identifier is then used to index into one of two Replacement Management Tables (RMTs)—an L2 Replacement Management Table and/or TLB Replacement Management Table—to control which sets within the L2 cache or TLB can be used by the access. This provides software with the ability to define an EA region in which data is continuously reused and should be retained or locked in the L2 cache and TLBs. Software can also define an EA region in which data is streamed with little to no reuse, such that only a small area of the L2 cache and TLB is used and the rest of the L2 cache and TLB entries are not replaced by reloads due to this streaming data.

The Address Range Start Registers also provide a real-mode or relocation-mode bit to specify whether the address match is a real address or a translated address. Storage accesses that do not match any range are assigned to replacement class 0. Eight replacement classes are supported (class 0 through 7).

For further details about the Address Range Registers, see *Section 6.3.1.3 Address Range Registers* on page 155.

Replacement Management Tables

The hypervisor assigns L2-cache sets and/or TLB sets to each class ID using the Replacement Management Tables, described in *Section 6.3 Replacement Management Tables* on page 154. The PPE MMU supports an 8-entry RMT for the L2 cache and an 8-entry RMT for the TLB. A 3-bit class ID is used to index into the table to select one of the RMT entries. Each entry of the RMT contains a replacement-enable bit for each way of the cache or TLB. For the L2 cache, the RMT is set up and controlled through the `L2_RMT_Setup` register and the `L2_ModeSetup1[RMT_mode]` bit. For the TLB, the RMT is setup and controlled through the `PPE_TLB_RMT` register. The `PPE_TLB_RMT` register is only effective when the hardware-managed TLB mode is enabled (`LPCR[TL]` = '0').

11.2.4 Controlling Interrupts and Environment

11.2.4.1 *Logical-Partition Environment Selector*

One of the bits in the Logical-Partition Environment Selector field, `LPCR[LPES]`, controls whether External Interrupts force `MSR[HV]` to '1' or leave it unmodified. This control is used by the hypervisor to determine whether External Interrupts are always delivered to the hypervisor or directly to the software being interrupted (the hypervisor or the operating system in the logical partition). Another `LPCR[LPES]` control bit is used to determine whether the Real Mode Offset facility is to be used by the logical partition, and to direct some of the interrupts to the hypervisor instead of the

software being interrupted—one bit for both purposes[23]. The System Reset, Machine Check, System Error, and Hypervisor Decrementer (HDEC) interrupts are always presented to the hypervisor.

11.2.4.2 ***External Interrupts***

External-Interrupt Handling

The hypervisor is typically responsible for managing External Interrupts (see *Table 9-1 PPE Interrupts* on page 239). This can be done in one of two ways:

- The logical partition supplies the First-Level Interrupt Handler (FLIH) and the interrupt vector in the Real-Mode Offset (RMO) region.
- The hypervisor supplies the FLIH and interrupt vector in the low real-address range or the Hypervisor Real-Mode Offset (HRMO) region.

In the first case, although the logical partition provides the FLIH, it typically will make hypervisor calls to determine the interrupting condition, mask interrupts, generate interrupts, and acknowledge interrupts. In either case, the hypervisor must retain direct control over the interrupt masks, interrupt status, interrupt routes, interrupt-generation registers, interrupt priorities, and interrupt-pending registers of the two Internal Interrupt Controllers (IICs), one for each PPE hardware thread, or of any External Interrupt controllers attached to the IOIF interfaces. All External Interrupts ultimately must be presented to one of the IICs. The IICs, in turn, provide the interrupt to the PPE thread.

Real-Time Processing of External Interrupts

In real-time applications, tasks must be completed within a specific period of time. In particular, interrupt processing for high-priority interrupts must be deterministic and worst-case latency of interrupt handling must be managed accordingly. In a logically partitioned system, the PPE may be currently assigned to one partition when a high-priority interrupt occurs that must be serviced by an operating system in another partition. Hypervisors traditionally defer External-Interrupt processing until the PPE is dispatched on the partition required to handle the External Interrupt. This can result in extremely long interrupt-processing delays, which are unacceptable to real-time applications.

Setting the LPCR[LPES] bit 0 = ‘0’ allows External Interrupts to be directly handled by the hypervisor, instead of by the operating system in the partition currently executing. However, the executing partition retains control over the MSR[EE] which enables or disables External Interrupts, with the potential to significantly delay the handling of a high-priority External Interrupt.

Mediated External Interrupts

To solve the potential problem described in the paragraph above, the PPE supports a Mediated External Exception mode (see *Section 9.5.7* on page 251 for details). This mode is enabled by a hardware implementation register (HID0[extr_hsrr] = ’1’). In this mode, with LPCR[LPES] bit 0 = ‘0’ to direct External Interrupts to the hypervisor, the MSR[EE] state does not inhibit External Interrupt delivery while the PPE is executing in non-hypervisor state. Instead of saving the interrupting

23. See *PowerPC Operating Environment Architecture, Book III* for details on the LPCR register.

PPE state in the SRR0 and SRR1 registers, the interrupting state is saved in HSRR0 and HSRR1 to avoid loss of context. In addition, the hypervisor would use the HSPRG registers for scratch registers instead of the SPRG registers. This provides the capability for the hypervisor to respond to the reception of a high-priority External Interrupt.

The hypervisor can preemptively swap the targeted PPE context to the partition that must service the interrupt. The hypervisor then generates a mediated external-exception request by setting the Mediated External-Exception Request (LPCR[MER]) bit to '1'. This mediated external exception condition honors the setting of the MSR[EE] state for the logical partition, and it will not provide a mediated External Interrupt to the partition until MSR[EE] = '1'. The occurrence of a Mediated External Interrupt saves the processor-element state in the SRR0 and SRR1 registers. A mediated external-exception request is outstanding when LPCR[MER] = '1', however the exception will never interrupt the processor element while in hypervisor mode (MER[HV] = '1' and MER[PR] = '0').

The Mediated External Interrupt facility is also available when enabled by HID0[extr_hsrr] = '1' and LPCR[LPES] bit 0 = '1'. In this mode, Direct External Exceptions are delivered as External Interrupts directly to the currently executing environment (hypervisor or the logical-partition software). However the interrupted processor-element state is saved in SRR0 and SRR1, and the interrupt handler would use the SPRG registers as scratch registers. The setting of MSR[EE] enables or disables the External Interrupt delivery in both hypervisor and the logical partition. The hypervisor can generate a Mediated External Exception and associated External Interrupt by using the LPCR[MER] bit. This is particularly useful, for example, when the hypervisor wishes to request the addition or removal of a resource for dynamic provisioning.

The LPCR[LPES] setting applies to both PPE threads, whereas the MSR[EE] and the LPCR[MER] are per thread.

For further details about interrupt handling, see *Section 9 PPE Interrupts* on page 237.

11.2.4.3 *Hypervisor Decrementer Interrupts*

In order to provide periodic maintenance tasks or timeshare resources among partitions, the hypervisor can use the Hypervisor Decrementer (HDEC). This decrementer will interrupt the processor element regardless of the MSR[EE] setting when MSR[HV] = '0'. An HDEC interrupt will not be delivered when MSR[EE]= '0' in hypervisor state. In addition, the LPCR register provides a Hypervisor Decrementer Interrupt Conditionally Enable bit (LPCR[HDICE]) to disable the hypervisor decrementer in all states.

11.2.4.4 *Machine Check Interrupts*

Only the hypervisor has the capability to enable or disable Machine Check interrupts by altering the MSR[ME] bit in the Machine State Register. This bit is used to control whether a subsequent machine check event causes a machine check interrupt or a system wide Checkstop.

11.2.4.5 *Time Base*

The Time Base (TB) register is readable in all privilege states. However, because the TB is used to manage system-wide time, it is only writable in the hypervisor state.

11.2.4.6 *Processor Identification*

The Processor Identification Register (`PIR`) is used by software to distinguish one PPE from another in a multi-CBE-processor system. It is typically used to identify communicators during communication between processor-elements and I/O devices. The register can be written only by the Power-On-Reset configuration bits, after which it is read-only. (For details about configuration bits, see the *Cell Broadband Engine Hardware Initialization Guide*.)

11.2.4.7 *Data Address Breakpoints*

The Data Address Breakpoint Register (`DABR`) and Data Address Breakpoint Register Extended (`DABRX`) are used for system-level debugging and can only be accessed by the hypervisor. This prevents unauthorized breakpoints being set in hypervisor data.

11.3 SPE Logical-Partitioning Facilities

The hypervisor can allocate SPEs among logical partitions. The hypervisor can also dynamically reassign an SPE from one partition to another. The SPEs can operate in a partition which does not—due to PPE-sharing—currently have a physical PPE assigned to it, until a PPE interrupt is generated by an SPE.

SPE software can only fetch instructions and load and store data directly from its LS using an LS address (LSA). SPE software has access to a limited set of communications and DMA facilities directly through the use of SPU channel instructions. Software in the SPU itself has no direct access to main storage. It must use the MFC DMA facilities to move data between the LS and the effective-address (EA) space. Each MFC contains an MMU, very similar to the MMU in the PPE, for EA-to-RA translation.

11.3.1 Access Privilege

Accesses to an SPE from all other system units (PPE, other SPEs, and I/O devices) is provided through MMIO registers. Therefore, with respect to logical partitions executing on the PPE, an SPE can be allocated to the logical partition by the hypervisor granting access to the SPE MMIO registers. These registers are divided into three groups, based on privilege, as specified in the *Cell Broadband Engine Architecture* document:

- *Privilege 1 Registers* (most privileged)—Used by the hypervisor to manage the SPE on behalf of a logical partition.
- *Privilege 2 Registers*—Used by the operating system in a logical partition to manage the SPE within the partition.
- *Privilege 3 Registers* (least privileged)—Used by problem-state (application) software, if direct access to the SPE from user space is supported by the operating system.

A hypervisor should not grant logical-partition access to the Privileged 1 group of SPE MMIO registers. These registers provide control over SPE memory management, interrupt management, versioning, SPE cache management, and SPE status and control.

11.3.2 Memory-Management Facilities

As with storage access by the PPE, the hypervisor must control storage access by an SPE allocated to a logical partition. The following hypervisor (Privilege 1) facilities are replicated for each SPE to provide this control:

- *MFC Logical Partition ID Register (`MFC_LPID`)*—Identifies the partition to which an SPE is assigned.
- *MFC State Register 1 (`MFC_SR1`)*—Controls MFC access to storage.
- *MFC TLB Management Registers (`MFC_SDR`, `MFC_TLB_Index_Hint`, `MFC_TLB_Index`, `MFC_TLB_VPN`, `MFC_TLB_RPN`, and `MFC_TLB_Invalidate_Entry`)*—Specify page-table properties and state.
- *MFC TLB Replacement Management Table Data Register (`MFC_TLB_RMT_Data`)*—Enables up to four Replacement Management Tables (RMTs).

The contents of the `MFC_SR1` register controls the following functions:

- Software- or hardware-managed MFC TLB reload on translation miss.
- SPU master run control.
- Relocation (translation) of MFC EAs to RAs.
- Problem- or privileged-state access by MFC.
- Ignore or honor PPE-issued **tlbie** instructions in the same logical partition.
- Aliasing of LS space into the main-storage EA space.

As in logical partitioning for the PPE, the hypervisor must control an MFC DMA controller's access to physical storage through the use of the hardware page table or TLBs, as described in *Section 11.2.3* on page 332. Unlike the PPE, the MFCs do not support a Real Mode Offset facility, because there is no operating system executing on an SPE and there is no requirement to access a low-memory area in real mode.

The hypervisor should not disable MFC Address Relocation for an SPE assigned to a logical partition, because this would provide uncontrolled access to all of the EA space. Relocation should only be disabled on SPEs that are allocated for hypervisor use, requiring real-mode access.

11.3.2.1 *Page Tables*

The hypervisor can use the `MFC_SR1` to specify that TLB (VA-to-RA) reloads should be handled either by a hardware lookup in a hashed page table in main storage or by direct software load of a TLB entry. When the hardware TLB reload is used, the MFC Storage Description register (`MFC_SDR`) must be set by the hypervisor to describe the hardware-accessible page table. This page table has the same format as the PPE page table. Hardware page tables can be common for all PPE and SPE resources in a logical partition, or they can be separate. When a common hardware page table is used, the `MFC_SR1` register should be set so that PPE-issued **tlbie** instructions are honored by all SPEs in the logical partition. In addition, the `MFC_LPID` register must be set by the hypervisor to the LPID for the logical partition that the SPE is assigned to. The SPE will ignore PPE-issued **tlbie** instructions if the PPE's LPID does not match the MFC LPID.

If per-SPE page tables are used, the `MFC_SR1` register should be set so that PPE-issued **tlbie** instructions are ignored by the SPEs in the logical partition.

11.3.2.2 *MFC TLB Management Registers*

The hypervisor can use the software TLB-reload mode on one or more of the SPEs by setting `MFC_SR1[TL]` = '1'. This can be useful in small-memory configurations to avoid the need for a large hardware-accessible page table in main storage. In this mode, when a page-translation miss occurs, the PPE is interrupted to load the TLB with the necessary translation and resume the operation. The TLB-management registers in the Privilege 1 area are used for this purpose.

When a miss occurs, the MFC TLB Index Hint (`MFC_TLB_Index_Hint`) register specifies which TLB congruency class needs to be loaded to resolve the miss. An LRU hint is also available in this register, indicating which TLB entry in the class is the least-recently used. The MFC TLB Invalidate Entry (`MFC_TLB_Invalidate_Entry`) register must be used to invalidate the TLB in the SPE before reloading a new one[24]. The MFC TLB Index (`MFC_TLB_Index`) register is used to specify which TLB entry is read or written by accesses to the MFC TLB VPN (`MFC_TLB_VPN`) and MFC TLB RPN (`MFC_TLB_RPN`) registers.

11.3.2.3 *TLB Entries*

Unlike the PPE TLB entries, the SPE TLB entries are not tagged with a logical partition ID[25], because the frequency of switching an SPE between logical partitions should be much lower than switching the PPE between logical partitions. Switching an SPE between logical partitions requires that all entries in the SPE TLB be invalidated when the SPE is switched from one partition to another. This is required because each partition manages its own VA space, therefore page translations are unique to the partition.

Like the PPE, in the hardware TLB-reload mode (`MFC_SR1[TL]` = '0') the MMU hardware uses a least-recently used (LRU algorithm) to determine which entry of the 4-way set-associate TLB will be replaced during a TLB reload. For finer control of this facility, the MFC MMU also supports a TLB Replacement Management Table (RMT), similar to that in the PPE MMU. This table is set up by the hypervisor for each SPE, using the MFC TLB Replacement Management Table Data Register (`MFC_TLB_RMT_Data`). This register has 4 fields, one for each replacement-class ID (RClassID), with each field containing four set-enable bits to indicate whether the set is to be used for the corresponding class. The RClassID for DMA commands is provided by the application program when queuing a DMA command.

11.3.2.4 *Access to Privileged Storage*

When assigning the SPE to a logical partition, the hypervisor can also specify whether the SPE is to have access to privileged storage as well as problem-state storage. This is specified by a the `MFC_SR1[PR]` bit. Storage-access privilege is specified by the storage key in the Segment Look Aside Buffers (SLBs) that the operating system controls in the logical partition, in conjunction with the storage-protection bits in the hardware page table or TLB entries that the operating system specifies and the hypervisor controls.

---

24. See *Section 4.3.5.3* on page 111 for details.
25. SPE TLB entries are invalidated by writing the MFC_TLB_Invalidate_Entry register, as opposed to the PPE method of issuing a **tlbie**, **tlbiel**, or **tlbia** instruction to invalidate PPE TLB entries.

11.3.2.5 *Aliasing LS to Main Storage*

In order to facilitate LS-to-LS DMA transfers, or the capability of an I/O device to access LS, the operating system can request the hypervisor to alias the allocated SPE's LS into the real-address space available to the logical partition or to the hypervisor. The enabling and disabling of LS aliasing is controlled by the `MFC_SR1[D]` bit.

11.3.3 Controlling Interrupts

The hypervisor has exclusive access to the SPE interrupt routing, interrupt mask and interrupt status registers, described in *Section 9.6.3 SPU and MFC Interrupts* on page 268. These registers are replicated for each SPE, and within each SPE there is one for each class of interrupt (error, translation, and application):

- Interrupt Routing Register (`INT_Route`).
- Interrupt Mask Registers (`INT_Mask_class0`, `INT_Mask_class1`, and `INT_Mask_class2`).
- Interrupt Status Registers (`INT_Stat_class0`, `INT_Stat_class1`, and `INT_Stat_class2`).
- SPU Identification Register (`SPU_IDR`).

By setting the Unit ID in the `INT_Route` register, SPE-initiated interrupts can be routed to a specific PPE thread or to an external interrupt controller attached to the IOIF0 or IOIF1 interface. This register allows each of the three classes of SPE initiated interrupts to be routed to the same or different units. This is useful if one unit is to handle error interrupts while a different unit is to handle translation or application interrupts.

As described in *Section 11.2.4* on page 337, when SPE interrupts (which are seen as External Interrupts) are routed to a PPE thread, the hypervisor can handle the interrupts directly or it can allow the operating system in the currently executing partition to handle the interrupt. In either case, the hypervisor should provide a set of SPE interrupt services to the logical partition. These services should include masking an allocated SPE interrupt, querying an SPE's interrupt status, and resetting an SPE interrupting condition.

11.3.4 Other SPE Management Facilities

The following additional facilities provide the hypervisor with per-SPE information and control:

- MFC and SPU Version Registers.
- MFC Cache Management Registers.
- MFC and SPU Error Management Registers.
- MFC and SPU Debug and Performance Monitor Registers.

11.3.4.1 *SPU Run Control*

The hypervisor can inhibit the operating system or an application in the logical partition from starting an SPU that has been allocated to it. This control is provided by the SPU Master Run Control bit, `MFC_SR1[S]`. It can also be used to stop a currently executing SPU that has been allocated. This can be useful in certain high-thermal conditions, or can be used to keep the SPU resources allocated to a logical partition for the purposes of retrieving the SPU context, but preventing additional tasks being started on the SPU. If an SPU was stopped with the `MFC_SR1[S]`

bit, the SPU will not resume execution until the bit has been enabled and a run request to the SPU Run Control (`SPU_RunCntl`) register has been initiated by the hypervisor, operating system, or application program.

11.3.4.2 *SPE Identification*

The SPU Identification Register (`SPU_IDR`) is set by hardware to a unique value for each physical SPE and can be used by the hypervisor to index per-SPE structures. The MFC and SPU Version Registers (`MFC_VR` and `SPU_VR`) are also set by hardware and are used by the hypervisor to manage the features and functions available for a given version. It is suggested that the hypervisor propagate this information to the software in the logical partitions as configuration information, as required.

11.3.4.3 *Cache Management*

Each SPE contains a small cache to support TLB loads from the hardware page table and the caching of atomic-update lock lines. The atomic cache can be flushed by the hypervisor using the MFC Atomic Flush (`MFC_Atomic_Flush`) register. This should always be done before putting an unallocated SPE into one of the low-power states (see *Section 15.1 Power Management* on page 423) because the contents of the cache can become stale or lost in these states. The cache uses real-address tags, so it is not necessary to flush the cache when switching the SPE between logical partitions.

11.3.4.4 *Error Management*

To support error detection, the hypervisor can use the SMM Hardware Implementation Dependent Register (`SMM_HID`) to enable SLB and TLB parity generation and checking, and to control the MFC performance, trace, and trigger settings. In addition, SPU illegal instruction and LS ECC error detection and correction functions can be enabled and controlled through the SPU ECC and Error Mask Registers (`SPU_ECC_Cntl`, `SPU_ECC_Stat`, `SPU_ECC_Addr`, `SPU_ERR_Mask`). In addition, a set of seven MFC Fault Isolation, Error Mask, Checkstop Enable Registers are provided.

If a DMA transfer is halted due to an illegal command, an SPU-initiated interrupt is generated and the hypervisor can access the MFC Command Error Register (`MFC_CER`) to determine which DMA command in the MFC queue caused the error.

11.3.4.5 *Debug and Performance Monitoring*

Several trace, trigger, and event control registers are provided to support debug and performance monitoring of all major logic blocks in the SPEs.

The hypervisor can enable DMA debug facilities through the use of the MFC address-compare facilities. The facilities are provided by the MFC Address Compare Control Register (`MFC_ACCR`), the MFC Local Storage Address Compare Register (`MFC_LSACR`), and the MFC Local Storage Compare Result Register (`MFC_LSCRR`). These registers are replicated for each SPE and allow halting of MFC DMA transfers when a specified LS address or main-storage page is read or written by a DMA transfer. The MFC Data Storage Interrupt Pointer Register (`MFC_DSIPR`) provides an index to the DMA command in the queue that triggered the address compare.

11.4 I/O-Address Translation

When a hypervisor creates logical partitions, it is important to restrict main-storage access by bus-mastering I/O devices (that is, I/O devices that can initiate DMA transfers to and from main storage), if the I/O devices are to be controlled directly by an operating system within the logical partition. Although a hypervisor can support I/O devices directly and it can fully virtualize them to software in the logical partition, this adds unacceptable latency and reduced throughput in many real-time or high-performance systems.

11.4.1 IOC Memory Management Units

The CBE processor's I/O Interface Controller (IOC) has an MMU that provides I/O-address translation and access protection for main-storage accesses initiated by I/O devices. The hypervisor must exercise full and exclusive control over the IOC's MMU to ensure that I/O devices supporting a logical partition cannot access resources in another logical partition. I/O devices access main storage through an I/O-address space that is separate from the real-address (RA) space.

The hypervisor can set up the IOC's MMU to support translation of I/O addresses to RAs. To do this, the hypervisor must enable I/O translation by setting the Enable bit of the I/O Segment Table Origin Register (`IOC_IOST_Origin[E]`) to '1' and the Translate Enable bit of the IOCmd Configuration Register (`IOC_IOCmd_Cfg[TE]`) to '1'. See *Section 7.4 I/O-Address Translation* on page 176 for details.

11.4.2 I/O Segment and Page Tables

When an I/O device initiates a DMA transfer, the IOC MMU looks the supplied I/O address up in an I/O segment table (IOST), then uses the segment table to look up the RA in an I/O page table (IOPT). Unlike PPE MMU and SPE MMU translations, I/O translations must be present in the IOST and/or IOPT cache entries before an I/O device attempts access. The CBE processor does not support the suspension and resumption of an I/O device's bus-mastering operation in order for the hypervisor to dynamically load IOST or IOPT entries for translations.

The hypervisor can enable the use of an IOST and IOPT in main storage—which the IOC MMU will access when an address is not found in the IOST or IOPT cache—by setting the `IOC_IOST_Origin[HW]` bit = '1'. If this option is not enabled, the hypervisor must directly manage the IOST and IOPT caches and set the `IOC_IOST_Origin[SW]` bit = '1'. Each IOST entry contains a Real Page Number (RPN) of the start of the IOPT for the segment, the number of 4-KB pages used for the segment's IOPT, the I/O page size used for all accesses in the segment, a Valid (V) bit and a Hint (H) bit. The Hint bit can be used by the hypervisor to keep frequently-used I/O-segment translations in the IOST cache.

When a valid IOST entry is found for an I/O address, bits in the I/O address are then used to index into the IOPT to look for the corresponding page-table entry (PTE). The PTE contains page-protection bits (restricting read and write access), a bit controlling cache coherency for the transfer, bits specifying storage-access ordering for the transfer, a Hint bit, an I/O Identifier (IOID), and the RPN corresponding to the RA to be used for the transfer. The IOID in each PTE prevents one I/O device from using another I/O device's address translation. This provides the necessary protection when I/O devices are allocated to different partitions. These IOIDs are normally assigned by the system supplier and are generated by the host bridges attached to the CBE I/O Interfaces.

If an I/O-device access does not find a valid IOST or IOPT entry, or if the access is prohibited by the IOPT protection bits, or if the IOID supplied by the device does not match the IOID in the IOPT, an I/O-device exception occurs and an External Interrupt is presented to the PPE.

11.5 Resource Allocation Management

The CBE's Resource Allocation Management (RAM) facility is described in *Section 8 Resource Allocation Management* on page 203. The hypervisor can apply the RAM facility to logical partitions. This facility can allocate specific amounts of memory or I/O bandwidth to the requesters (PPE, SPEs, or I/O devices) in each logical partition. The combined capabilities of logical partitioning and resource allocation management provide a level of resource control that is not typically found in conventional logical-partitioning processors.

11.5.1 Combining Logical Partitions with Resource Allocation

Conventional logical-partitioning processors divide up real memory, processing units, and I/O devices among logical partitions. This works well in most environments, but when one or more operating environments in the logical partitions must meet real-time deadlines, this type of resource allocation is insufficient.

For example, if partition 1 is allocated 70% of the PPE and five of the eight SPEs, and partition 2 is allocated 30% of the PPE and three of the eight SPEs, it is possible for applications in partition 2 to use so much main-storage bandwidth or I/O bandwidth that logical partition 1, which has more processing units, cannot meet its real-time schedule because the remaining memory or I/O bandwidth is insufficient. To alleviate this problem, a CBE hypervisor can use the EIB Resource Allocation Facility to manage the amount of real memory and I/O capacity the units allocated to a logical partition receive.

11.5.2 Resource Allocation Groups and the Token Manager

In the CBE processor, the requesters assigned to each logical partition can be assigned to one of four managed Resource Allocation Groups (RAGs). The hardware supports a Token Manager that generates tokens for the managed resources at hypervisor-programmed rates for each RAG. *Section 8.5* on page 212 describes the details.

Typically, the system supplier provides configuration information to the hypervisor in order for the hypervisor to determine maximum usable capacity of the managed resources. The hypervisor uses this information to ensure that the token-generation rate for the resource in all concurrently active RAGs does not exceed the maximum capacity of the resource.

Each managed resource also provides a feedback mechanism to the Token Manager, indicating that it is becoming overcommitted. The hypervisor can enable an External Interrupt to the PPE when a resource becomes overcommitted. The hypervisor programs queue-threshold levels in the IOC units (for IOIF0 and IOIF1) and in the Memory Interface Controller (MIC) for the DRAM memory banks.

The hypervisor uses these methods to assign resources to managed logical partitions:

- *PPE*—When the hypervisor assigns the PPE to a managed logical partition, the hypervisor enables the RAG requestor and sets a Resource Allocation Identifier (RAID) for that logical partition using the BIU Mode Setup Register 1 (`BIU_ModeSetup1`).

- *SPE*—When the hypervisor assigns an SPE to a managed logical partition, the hypervisor enables the RAG requestor and sets an RA Identifier for that logical partition using the `RA_Enable` register and the `RA_Group_ID` register.
- *I/O*—When the hypervisor assigns an IOIF unit to a managed logical partition, the hypervisor enables the RAG requestor and sets a virtual channel for outbound transfer from that logical partition using the `IOC_IOCmd_Cfg` register.

For complete details, see *Section 8 Resource Allocation Management* on page 203.

11.6 Power Management

The CBE processor can support multiple power management states, as described in *Section 15 Power and Thermal Management* on page 423. Many of the units in the CBE processor support dynamic power management when the units are not being used, and this requires no explicit action on the part of software. However, the CBE processor does support explicit power management for the low-power states—the MFC Pause, SPU Pause, SPE State Retained and Isolated (SRI), PPE Pause (0), and CBE Slow states.

11.6.1 Entering Low-Power States

When multiple logical partitions are present, only the hypervisor can determine if conditions are right to enter or exit these low-power states. Under certain system-specified conditions, it may be advisable to slow the CBE Core Clock (`NClk`) frequency, as described in *Section 13 Time Base and Decrementers* on page 375.

When a logical partition yields both of its PPE threads, and no other logical partitions are dispatchable (all waiting on external events), the hypervisor may elect to enable the PPE Pause (0) state and suspend each thread. Prior to entering this state, the hypervisor must write a PPE control register to designate the conditions under which the PPE Pause (0) state is to be exited. Machine Check, System Error, and Thermal interrupts must always exit the PPE Pause (0) state. The hypervisor can conditionally enable resumption on a Decrementer event or on an External Interrupt.

A low-power, system-specific hibernation state may or may not be supported by the system (it is not specifically supported by the CBEA). If hibernation state is supported by the system, it requires support from the system controller and system designer, as well as the hypervisor and in many cases the I/O devices. Hibernation mode consists of saving all volatile state into system DRAM, switching the DRAM to self-timed refresh mode and disabling the power to the CBE processor. It also requires that an 8 KB block of contiguous physical DRAM be left unused by system firmware or software, and dedicated to the Memory Interface Controller (MIC) recalibration-pattern buffer.

11.6.2 Thread State Suspension and Resumption

Unlike previous PowerPC processing units, the PPE retains the full hardware thread state when the thread is suspended. When the thread is resumed, it starts executing from the reset vector, and the reset interrupt handler can choose to resume the execution at the instruction following the instruction that suspended the thread.

When the hypervisor has determined that an SPE is no longer being used by a logical partition (it has freed the resource), the SPE can be held in a low-power SPU Pause state, with the SPU stopped and the MFC suspended. If the hypervisor determines that the SPE will not be used for a relatively long period (several seconds or minutes), the hypervisor can place the SPU in the SPE State Retained and Isolated (SRI) state.

11.7 Fault Isolation

Many faults detected by the CBE processor, including programming errors related to hypervisor resources, are non-recoverable and indicate a significant failure that could compromise data integrity. These faults result in a CBE Checkstop (see *Section 9.5.2* on page 246) which immediately halts all CBE processing elements and must be handled or recorded by a system controller external to the CBE processor.

However, some faults are recovered by the CBE-processor hardware, are software recoverable, or may be caused by software errors in the hypervisor or logical partition. Faults detected while the PPE is accessing a resource generate a Machine Check interrupt (also *Section 9.5.2* on page 246) that must be handled by the hypervisor. Some faults detected by the SPE generate a System Error interrupt (also *Section 9.5.16* on page 257) that also must be handled by the hypervisor or logical partition.

11.8 Code Sample

The following code sample illustrates one method of making an ABI-compliant hypervisor call. In particular, it demonstrates making a example (H_EX) request that accepts two input parameters (`arg1` and `arg2`) and returns two resulting words. The hypervisor call is invoked by executing the system call instruction (**sc**) with the LEV field of ‘1’. The hypervisor request is specified in register 3.

11.8.1 Error Codes and Hypervisor-Call (hcall) Tokens

An example set of error codes and Hypervisor call (hcall) tokens:

```
#define H_Success;        0
#define H_Hardware;      -1 /* Hardware error */
#define H_Function;      -2 /* Not supported */
#define H_Privilege;     -3 /* Caller not in privileged mode */
#define H_Parameter;     -4 /* Partition violation/conflict. */
```

11.8.2 C Functions for PowerPC 64-bit ELF Hypervisor Call

Examples of creating C-callable functions that perform a Hypervisor call for PowerPC 64-bit ELF:

```
filename: hcall_ex.s
        .set     H_EX, 0x104     # example hcall token
        .section ".opd","aw"
        .align 3
```

```
        .globl hcall_ex
hcall_ex:
        .quad .hcall_ex,.TOC.@tocbase, 0
        .previous
        .size hcall_ex,24
        .section ".text"
        .align 2
        .globl .hcall_ex
.hcall_ex:
        std     r3,-8(1)   # r3 (array of values) stored in stack
        li      r3,H_EX    # load r3 with hypervisor code
        sc 1               # Hypervisor Trap
        ld      r12,-8(1)  # reload array into r12
        cmpi    0,12,0     # only store return regs if array is non-NULL
        bne     ret2       # this &hcall; only returns contents of r4,r5
        blr                # return no values

ret8:   std r11,(7 * 8)(r12)
ret7:   std r10,(6 * 8)(r12)
ret6:   std r9,(5 * 8)(r12)
ret5:   std r8,(4 * 8)(r12)
ret4:   std r7,(3 * 8)(r12)
ret3:   std r6,(2 * 8)(r12)
ret2:   std r5,(1 * 8)(r12)
ret1:   std r4,(0 * 8)(r12)
        blr
```

Example of how to call the assembler code above:

```
filename: ex.c
void example(unsigned long long arg1, unsigned long arg2)
{
        long  rc;
        unsigned long results[2];

        rc = hcall_ex(results, arg1, arg2);
        if (rc != H_Success) {
                ... Failure Case ...
        }
        ...
}
```

12. SPE Context Switching

12.1 Introduction

The ability to save and restore an SPE's context is important for two primary reasons: preemptive or cooperative context-switching SPE tasks (programs) for virtualization of the SPEs, and debugger access and interaction with an SPE's execution state.

Saving and restoring an SPE context can be very expensive in terms of both time and system resources. The complete SPE context consists of:

- 256 KB of LS.
- 128, 128-bit General Purpose Registers (GPRs).
- Special Purpose Registers (SPRs).
- Interrupt masks.
- MFC-command queues.
- SPE channel counts and data.
- MFC Synergistic Memory Management (SMM) state.
- Other privileged state.

In all, the SPE context occupies more than 258 KB of storage space.

A programming model that allocates SPEs in a serially reusable manner—in which an SPE is assigned a task until it completes, then it is assigned another task—generally results in more efficient use of SPEs than does a programming model that implements frequent context switches. The serially reusable model requires fewer context switches, and less context needs to be saved and restored during a context switch.

A *preemptive context switch* facilitated by privileged software on the PPE is the most costly form of context save and restore, because the context is not known and a worst-case save and restore of all context state must be performed. An *application-yielding* context switch, in which the application using the SPE determines the timing and amount of context to be saved and restored, can minimize the overhead in terms of cycles and space used to save and restore context.

Note: *If an I/O device interfaces directly to an SPE, the SPE's physically mapped LS interacts with that device. Because the LS is part of the context that is switched, preemptive context switching of an SPE that interfaces directly with an I/O device should be avoided, unless the I/O device can first be reliably quiesced.*

Several machine resources must be saved and restored when switching an SPE's context. This chapter provides a basic overview of resources and save-restore sequences for a preemptive context switch. The sample sequences assume the SPE is already stopped at context-save time. Some operating environments can forgo restoring the context of an SPE that has been stopped, especially when stopped on error conditions.

12.2 Data Structures

The data that must be saved in order to suspend and later resume execution of an SPE task is stored in two related structures. One is in the LS of the SPE being saved or restored; the other is in main storage:

- *Local Store Context Save Area (LSCSA)*—A data structure in the SPE's Local Store (LS). It contains the local problem state that the SPE itself saves to and restores from during a context switch. The LSCSA is a temporary data structure that is copied to/from the permanent copy in the CSA.
- *Context Save Area (CSA)*—A data structure in main storage that contains all of the SPE state, both problem state and privileged state. The CSA includes a copy of the LSCSA.

12.2.1 Local Store Context Save Area

The LSCSA is the area of LS in which the SPU temporarily stores local problem state—primarily the GPRs, SPRs, and channel state. This is by far the smaller of the two data structures, requiring only a few KB of space.

The SPU saves to the LSCSA as part of a code sequence during a context-save, and it restores from the LSCSA as part of another code sequence during a context-restore. During a context-save, after saving to its LSCSA in the LS, the SPU then saves the LSCSA (along with the high 240 KB of the LS) to the CSA in main storage. Likewise, during a context-restore, the SPU first loads the LSCSA (along with the high 240 KB of the LS) into the LS from the CSA in main storage, and then the SPU restores its own local state from the LSCSA in LS.

Due to DMA-alignment restrictions, the LSCSA memory region should, at a minimum, be aligned to a 128-byte boundary. For many environments, it may be more practical to align the region to a page boundary.

12.2.2 Context Save Area

The CSA is a main-storage structure that contains the entire SPE context state, including the 256 KB LS, the 128 GPRs, the SPRs, the state of all status registers, interrupt masks, MFC-command queues, SPE channel counts and data, MFC SMM state, and other privileged state. During save and restore sequences, small parts of the CSA are written and read by the PPE, and large parts are written and read by the SPE.

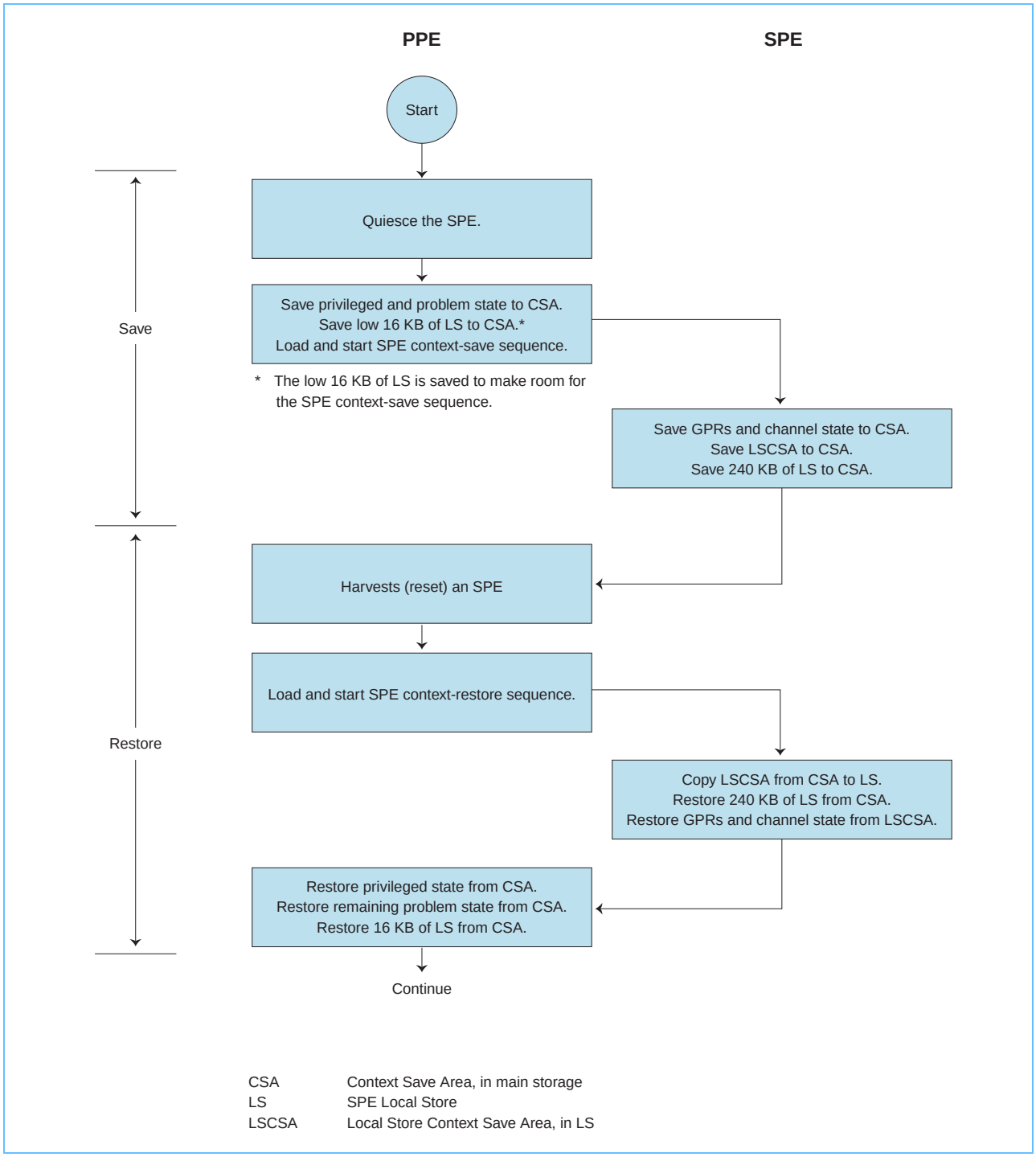
It may be best to order the CSA data so that the save areas for the LSCSA copy come first, followed by the save area for the LS. Otherwise, the SPE context-switch code may need to load immediate values for offsets that exceed the size of the LS, and thus require multiple assembly instructions to code.

12.3 Overview of SPE Context-Switch Sequence

This section gives an overview of the steps necessary for a preemptive context switch of an SPE task. A preemptive context switch consists of the combination of a context-save (*Section 12.3.1* on page 354) followed by a context-restore (*Section 12.3.2* on page 354). A full, detailed sample sequence is provided in *Section 12.5* on page 359.

The entire SPE context is not accessible from the PPE. As such a SPE context save or restore requires the assistance of specialized SPE code. A pictorial overview of the SPE context switch sequence is provided in *Figure 12-1* on page 353.

Figure 12-1. SPE Context-Switch Sequence



12.3.1 Save SPE Context

1. *Quiesce the SPE* (performed by the PPE):
   a. Disable SPE interrupts.
   b. Disable problem-space access to this SPE (unmap pages).
   c. Disable MFC command queues.
   d. Disable SPU execution.
   e. Disable all access to this SPE from other SPEs (unmap pages).
   f. Handle pending interrupts from this SPE.
2. *Save Privileged and Critical Problem State in CSA, Load and initiate SPU Context-Save Sequence on the SPU* (performed by the PPE):
   a. Save MFC command state, including command queues.
   b. Save privileged state, including SLB entries, resource allocation masks, and so forth.
   c. Save problem state, including mailbox data and SPE channel counts.
   d. Purge the MFC command queues.
   e. Resume MFC command processing.
   f. Invalidate MFC SLB entries, and reload with SLB entry for loading the SPU Context-Save Sequence code (Step 3, below).
   g. Invalidate MFC TLB entries.
   h. Reset interrupts.
   i. Save the low 16 KB of LS in the CSA.
   j. Place CSA effective-address pointer in SPU Signal Notification registers.
   k. Copy the SPU Context-Save Sequence code into the low 16 KB of LS, and start its execution.
3. *SPU Context-Save Sequence* (performed by the SPE):
   a. Save the low 16 GPRs in the LSCSA.
   b. Read SPU Signal Notification registers to obtain effective address of the CSA.
   c. Save SPU accessible channel state in the LSCSA.
   d. Save the high 240 KB of LS in the CSA.
   e. Save the high 112 GPRs in the LSCSA.
   f. Save the LSCSA in the CSA.

12.3.2 Restore SPE Context

1. *Harvest (reset the state of) an SPE* (performed by the PPE):
   a. Disable SPE interrupts.
   b. Disable problem-space access to this SPE (unmap pages).
   c. Disable all access to this SPE from other SPEs (unmap pages).
   d. Disable MFC command queues.

   e. Disable SPE execution.
   f. Invalidate MFC TLB entries.
   g. Handle pending interrupts from this SPE.
   h. Purge MFC command queues.
   i. Invalidate MFC SLB entries.
   j. Reset SPE channel state.
   k. Reset interrupts.

2. *Load and initiate SPU Context-Restore Sequence on the SPU* (performed by the PPE):
   a. Invalidate MFC SLB entries, and reload with entry for SPU Context-Restore Sequence code access.
   b. Copy mailbox state from the CSA to the CSA's copy of the LSCSA.
   c. Place CSA effective-address pointer in SPU Signal Notification registers.
   d. Copy the SPU Context-Restore Sequence code into the low 16 KB of LS, and start execution.
3. *SPU Context-Restore Sequence* (performed by the SPE):
   e. Read SPU Signal Notification registers to obtain effective address of the CSA.
   a. Restore the high 240 KB of the LS from the CSA.
   b. Restore the high 112 GPRs from the LSCSA.
   c. Restore channel and other problem state from the LSCSA.
   d. Restore low 16 GPRs from the LSCSA.
4. *Restore Privileged and Remaining Problem State* (performed by the PPE):
   a. Restore SPU run control state.
   b. Restore low 16 KB of LS from the CSA.
   c. Restore interrupt state.
   d. Restore MFC command queues from CSA.
   e. Restore privileged state from the CSA.
   f. Restore additional problem state from the CSA.
   g. Restore MFC SLB entries from the CSA.
   h. Enable SPU execution.
   i. Restore MFC control state from the CSA.
   j. Enable interrupts.

12.4 Implementation Considerations

12.4.1 Locking

Due to the length of the context-switch sequence, it may be appropriate to use semaphores rather than spin locks for access-locking. This allows other operations (for example, context-switch on other SPEs) to proceed when multiple threads are attempting to exclusively access the same SPE during context switch.

12.4.2 Watchdog Timers

Non-real-time systems may not be able to set reasonable time-out values for the entire context-switch sequence. This is due to the potential for paging activity, unexpected external events (for example, network), and so forth. It may be more appropriate to set timers on individual operations—for example, waiting for a purge of MFC command queues to finish.

12.4.3 Waiting for Events

Non-real-time systems may suffer significantly from excessive busy-polling for SPE context-switch events. It may be more appropriate to yield or sleep on events, rather than busy-poll. Enabling handlers for SPE class2 interrupts (stop-and-signal, tag-group completion) may be useful. See *Section 9.6.3 SPU and MFC Interrupts* on page 268 for details. For a description of SPE events, see *Section 18 SPE Events* on page 463.

12.4.4 PPE's SPU Channel Access Facility

Privileged PPE software has access to the four event-management channels and the hidden Pending Event register (*Section 18.2 Events and Event-Management Channels* on page 464) through the PPE's SPU Channel Access Facility. The SPU Channel Access Facility can initialize, save, and restore the SPU channels.

The facility consists of three MMIO registers:

- SPU Channel Index Register (`SPU_ChnlIndex`).
- SPU Channel Count Register (`SPU_ChnlCnt`).
- SPU Channel Data Register (`SPU_ChnlData`).

The SPU Channel Index Register is a pointer to the channel whose count and data is accessed by the SPU Channel Count Register and SPU Channel Data Register, respectively. See the *Cell Broadband Engine Architecture* for details about the PPE's SPU Channel Access Facility.

12.4.5 SPE Interrupts

It is possible to perform a context switch with SPE interrupts disabled. In this case, memory regions containing the context-switch program or data should be pinned (locked), and the MFC's SMM should be pre-loaded with the SLB and TLB entries necessary for these memory regions. Polling of MMIO registers is necessary in order to determine the completion status for data transfers, SPE execution, and so forth.

Non-real-time environments may prefer to enable interrupt handlers for address-translation (SPE class1) interrupts, so that code or data can be accessed as virtual memory rather than pinned real memory. Handlers for error (SPE class0) interrupts and application (SPE class2) interrupts can be used to detect potential errors or completion events (for example, tag-group complete, stop-and-signal). See *Section 9.6.3 SPU and MFC Interrupts* on page 268 for details.

12.4.6 Suspending the MFC DMA Queue

Depending on the OS, address-translation (SPE class1) interrupts might be processed by a deferred interrupt handler. The use of deferred interrupt handling complicates the context-save operation in the way that the MFC command queues are suspended and saved.

In this case, it is not enough for the context-save code to simply disable SPE interrupts. It is also necessary to prevent a deferred interrupt handler from restarting an MFC command after the queue has been suspended. This may be done by setting a per-SPE save-pending flag, indicating to the deferred handler that the MFC command should not be restarted.

12.4.7 SPE Context-Save Sequence and Context-Restore Sequence Code

The SPE's Context-Save Sequence and Context-Restore Sequence code should probably be written in SPE assembly language, so that register and memory use can be carefully controlled. The sequence described in *Section 12.5* on page 359 assume the code is written to execute in a portion of the low 16 KB of LS and uses only the first 16 GPRs.

A straight-forward implementation of either the save or restore sequence will typically use less than 1 KB for the program text. Because the save and restore sequences have much in common, it is possible to write a combined switch program that fits into roughly the same space. A further 2 KB will be used at run-time for saving or restoring registers.

Special **stop** instruction codes should be reserved for indicating the successful completion of a save and a restore.

12.4.8 SPE Parameter Passing

The only parameter needed by the save and restore code is the location of the CSA region in main storage. This may be communicated either by having privileged software running on the PPE store this address to a known location in LS before starting the SPE, or by writing this 64-bit address to the 32-bit SPU Signal Notification 1 and 2 registers as is done in *Section 12.5* on page 359.

12.4.9 Storage for SPE Context-Save Sequence and Context-Restore Sequence Code

The SPE context-switch code is trusted by the OS and must therefore reside in privileged memory. The code must be copied by the PPE from main storage into LS, so it should be, at a minimum, aligned to a 128-byte boundary for efficient transfer by the MFC. It may be possible to dump the SPE context-switch object code into raw hexadecimal form, so that it can be initialized and aligned in memory at compile time, rather than run time.

12.4.10 Harvesting an SPE

There may be situations in which an SPE context is not immediately restored to the SPE on which it was originally executing. This may be the case when the context is executing under control of a debugger. There may also be conditions that cause the SPE context-save operation to fail (for example, a bad address translation for a save to the CSA in the user-memory region).

In such cases, this or another SPE needs to be harvested. Harvesting means resetting (clearing) the state of an SPE, and it allow a subsequent restore operation to be performed to that SPE. The steps in the context-restore operation in *Section 12.5.2.1* on page 365 describe how an SPE can be harvested.

12.4.11 Scheduling

Sharing SPEs, as opposed to serially reusing SPEs, can be very expensive. A fully preemptive SPE context switch may take many μsecs. It may, however, still be desirable for an OS to share SPEs among multiple user programs. What an individual SPE gives up in terms of context-switch overhead, the CBE processor can make up for in the number of available SPEs. Taken together, overall system interactivity could be considered reasonable, even if the latency for an individual SPE switch is somewhat high.

To avoid thrashing, the SPE-switch time should be factored into the time-slice quantum that is assigned to an SPE context. Typically, shorter quanta are used for interactive processing, and longer quanta are used for batch processing. Selection of time-slice quanta is a always compromise between these two processing goals. The best approach is to use a quantum that is short enough to seem interactive, but long enough to allow forward progress. A good rule of thumb for minimum quanta on a more traditional processor might be 10 times the switch-time. However, given the abundance of SPEs and relatively long switch time, it is probably best to err on the side of longer rather than shorter quanta.

12.4.12 Light-Weight SPE Context Save

The sequence outlined in *Section 12.5* on page 359 describes a fully preemptive SPE context switch. In certain environments, however, it may be desirable to support light-weight context switching. This could be implemented in a cooperative manner by having the PPE send a privileged attention event to a program executing on an SPE. If the SPE program responds to the event within a fixed time-out—for example, by executing the stop and signal (**stop**) instruction with special return code—then the PPE might perform a simplified context-save sequence. Otherwise, the PPE would perform a full context save, and might subsequently penalize that program by lowering its priority within the system.

To implement such a policy, there would be some cooperation between the OS and SPE application code. For example, upon receipt of the privileged attention event, the SPE program might wait for all pending MFC commands to complete before executing the **stop** instruction. This would allow the PPE to avoid suspending and saving the MFC command queues.

Another alternative would be to support reentrant contexts that are never saved, provided that they respond to the privileged attention event within a fixed time-out period. There are several variations possible between a fully preemptive switch and reentrant contexts.

12.5 Detailed Steps for SPE Context Switch

Section 12.3 on page 352 gave an overview of the sequence of steps involved in preemptively saving and restoring an SPE context. The sequences in this section expand on that overview by providing more detail.

These sequences depict the saving and restoring of SPE context by an OS that is running at both Privilege 1 and Privilege 2 levels—that is, an operating environment without a separate hypervisor and without support for logical partitioning (see *Section 12.6 Considerations for Hypervisors* on page 373). The sequences also assume that the OS uses hardware-managed TLBs and a common hardware-managed page table, and that it has allocated an area of main storage for the CSA, as described in *Section 12.2 Data Structures* on page 352.

Although the sequences handle preemptive saving and restoring, they can also handle cases in which the SPE is already stopped at context-save time or has stopped itself after context-save read its status. Some operating environments may forego restoring the context of an SPE that was not running at context-save time (especially when stopped on error conditions). However, support for restoring is included in this example for completeness.

The sequences do not include saving and restoring performance-monitor and trace registers or application-debug facilities. Interaction between the context-switching system and the debug and performance-monitor subsystems dictate how these registers are dealt with during context switch.

12.5.1 Context-Save Sequence

12.5.1.1 *PPE Context-Save Sequence for SPE*

This section outlines how to fully save the context of an active SPE:

1. Acquire a software SPE mutual-exclusion lock (if multithreading is being used).
2. If any `SPU_Status[E, L, IS]` field is set to ‘1’, this SPE is in the isolate state and cannot be context-saved at this time:
   - If multithreading is being used, release software SPE mutual-exclusion lock.
   - If the isolate state is an unexpected condition, harvest an SPE by going to Step 1 of the PPE Context-Restore Sequence on page 365.
   - If the isolate state is allowed, exit the context-save sequence.
3. Disable all interrupts (class 0, 1, and 2) from this SPE:
   a. Save the `INT_Mask_class0` register in the CSA. Then, write the `INT_Mask_class0` register with ‘0’.
   b. Save the `INT_Mask_class1` register in the CSA. Then, write the `INT_Mask_class1` register with ‘0’.
   c. Save the `INT_Mask_class2` register in the CSA. Then, write the `INT_Mask_class2` register with ‘0’.
4. Set a software watchdog timer, specifying the maximum time for a context-save sequence.
5. Inhibit problem-space access to this SPE, if provided, by unmapping the virtual pages assigned to the SPE MMIO problem space.

6. Read the SPU_Status register again. If any SPU_Status[E, L, IS] fields are active, this SPE is isolated and cannot be context-saved at this time. If the isolate state is an expected condition, the context-save sequence should be exited by skipping to Step 74 on page 372. If this is an unexpected condition, skip to Step 4 on page 365 to reclaim this SPE and restore a context to it.
7. Set a software Context Switch Pending flag.
8. Read the MFC_CNTL[Ss] register field:
   - If MFC_CNTL[Ss] is set to '01', poll MFC_CNTL[Ss] until it is set to '11'. Then, save the MFC_CNTL register in the CSA with MFC_CNTL[Sc] set to '1' and MFC_CNTL[Sm] set to '1'.
   - If MFC_CNTL[Ss] is set to '00', set MFC_CNTL[Sc] to '1'. Then, poll MFC_CNTL[Ss] until it is set to '11', and save the MFC_CNTL register in the CSA with MFC_CNTL[Sc] set to '0' and MFC_CNTL[Sm] set to '0'.
   - If MFC_CNTL[Ss] is set to '11', save the MFC_CNTL register in the CSA with MFC_CNTL[Sc] set to '1' and MFC_CNTL[Sm] set to '1'.
9. Save the SPU_RunCntl register in the CSA. This value contains the Application Desired State.
10. Save the MFC_SR1 register in the CSA.
11. Read the SPU_Status[R] register field:
    - If SPU_Status[R] is set to '0', save the SPU_Status register in the CSA.
    - If SPU_Status[R] is set to '1':
      a. Set SPU_RunCntl[Run] to '00'.
      b. Issue a PPE **eieio** instruction.
      c. Poll SPU_Status[R] until it is set to '0'. If SPU_Status[I,S,H, and P] are all set to '0', write the SPU_Status register in the CSA with the R bit set to '1'. Otherwise, write the SPU_Status register in the CSA with the R bit reset to '0'.
12. Read the MFC_CNTL[Ds] register field. Then, update the saved copy of this field in the CSA.
13. Write the MFC_CNTL[Dh] register field to '1' to halt the decrementer.
14. Read the special purpose Time Base (TB) register and save it in the CSA.
15. Prevent other SPE accesses to this SPE by unmapping this SPE's pages from their address spaces.
16. Write the MFC_MSSync register. Then, poll MFC_MSSync[P] for a value of '0'.
17. Write all of the MFC_TLB_Invalidate_Entry[IS,VPN,L,Lp] register fields to '0'. Then, issue a PPE **sync** instruction.
18. Handle any pending interrupts from this SPE. This is OS- or hypervisor-specific. One option is to re-enable interrupts to handle any pending interrupts, with the interrupt handlers recognizing the software Context Switch Pending flag, to ensure that SPE execution or the MFC command queue is not resumed.
19. If the MFC_CNTL[Q] register field is set to '0' (MFC command queues not empty), then save 96 doublewords from the MFC_CQ_SR registers in the CSA.
20. Save the Prxy_QueryMask register in the CSA.
21. Save the Prxy_QueryType register in the CSA.
22. Save the MFC_CSR_TSQ register in the CSA.

23. Save the `MFC_CSR_Cmd1` and `MFC_CSR_Cmd2` registers in the CSA.
24. Save the `MFC_CSR_ATO` register in the CSA.
25. Save the `MFC_TClassID` register and in the CSA.
26. Write the `MFC_TClassID` register with the value x'10000000' (T0 Quota = 16).
27. Write the `MFC_CNTL[Pc]` register field to '1' (purge queue).
28. Poll the `MFC_CNTL[Ps]` register field until the value '11' is read (purge complete).
29. If the `MFC_SR1[R]` register field is set to '1':
    a. Save the `SLB_Index` register in the CSA.
    b. For index = 0 through 7:
        1. Write the index value to `SLB_Index`. Then, issue a PPE **eieio** instruction.
        2. Save the `SLB_ESID` register in the CSA.
        3. Save the `SLB_VSID` register in the CSA.
30. Write the `MFC_SR1` register with `MFC_SR1[D]` set to '0', `MFC_SR1[S]` set to '1', and `MFC_SR1[TL,R,PR, and T]` set correctly for the OS-specific environment.
31. Save the `SPU_NPC` register in the CSA.
32. Save the `SPU_PrivCntl` register in the CSA.
33. Write the `SPU_PrivCntl[S,Le,A]` register field to '0'.
34. Save the `SPU_LSLR` register in the CSA.
35. Set the `SPU_LSLR` register to the size of implemented LS, minus one.
36. Save the `SPU_Cfg` register in the CSA.
37. Save the `PM_Trace_Tag_Wait_Mask` register in the CSA.
38. Save the `RA_Group_ID` and `RA_Enable` registers in the CSA.
39. For the following channels:
    - SPU_RdEventStat (channel index 0).
    - SPU_WrEventMask (channel index 1).
    - SPU_RdSigNotify1 (channel index 3).
    - SPU_RdSigNotify2 (channel index 4).
    - MFC_RdTagStat (channel index 24).
    - MFC_RdListStallStat (channel index 25).
    - MFC_RdAtomicStat (channel index 27).
        a. Write the channel index to the `SPU_ChnlIndex` register. Then, issue a PPE **eieio** instruction.
        b. Save the channel data from `SPU_ChnlData` register in the CSA.
        c. Save the channel count from `SPU_ChnlCnt` register in the CSA.
        d. Write '0' to the `SPU_ChnlData` register.
        e. Write '0' to the `SPU_ChnlCnt` register.
40. Save the `SPU_Mbox_Stat` register in the CSA.

41. Save the `SPU_Out_Mbox` register in the CSA.
42. Save the `SPU_Out_Intr_Mbox` register in the CSA.
43. For the `SPU_RdInMbox` channel:
    a. Write the channel index value 29 to the `SPU_ChnlIndex` register. Then, issue a PPE **eieio** instruction.
    b. Save the `SPU_ChnlCnt` register in the CSA.
    c. Save the `SPU_ChnlData` register in the CSA's `SPU_RdInMbox` data0 area.
    d. Save the `SPU_ChnlData` register in the CSA's `SPU_RdInMbox` data1 area.
    e. Save the `SPU_ChnlData` register in the CSA's `SPU_RdInMbox` data2 area.
    f. Save the `SPU_ChnlData` register in the CSA's `SPU_RdInMbox` data3 area.
44. For the `MFC_Cmd` channel:
    a. Write the channel index value 21 to the `SPU_ChnlIndex` register. Then, issue a PPE **eieio** instruction.
    b. Save the `SPU_ChnlCnt` register in the CSA.
45. For the following channels:
    - `MFC_Cmd` (channel index 21, count = 16).
    - `MFC_WrTagUpdate` (channel index 23, count = 1).
    - `SPU_WrOutMbox` (channel index 28, count = 1).
    - `SPU_WrOutIntrMbox` (channel index 30, count = 1).
      a. Write the channel index to the `SPU_ChnlIndex` register. Then, issue a PPE **eieio** instruction.
      b. Write the count to the `SPU_ChnlCnt` register.
46. Write the `MFC_CNTL[Sc]` register field to '0' and `MFC_CNTL[Sm]` to '0' (resume queue processing).
47. If the `MFC_SR1[R]` register field is set to '1', write '0' to `SLB_Invalidate_All` register. Then:
    a. Write x'0' to the `SLB_Index` register.
    b. Write the `SLB_VSID` register with the VSID to the SPU Context-Save Sequence code.
    c. Write the `SLB_ESID` register with the ESID to the SPU Context-Save Sequence code.
    d. Write x'1' to the `SLB_Index` register.
    e. Write the `SLB_VSID` register with the VSID to the CSA.
    f. Write the `SLB_ESID` register with the ESID to the CSA.
48. Change the software Context Switch Pending flag to Context Switch Active.
49. If the OS is using interrupts instead of polling:
    a. Write the `INT_Stat_class0` register with -1 to reset all class 0 interrupts.
    b. Write the `INT_Stat_class1` register with -1 to reset all class 1 interrupts.
    c. Write the `INT_Stat_class2` register with -1 to reset all class 2 interrupts.
    d. Write the `INT_Mask_class0` register with an OS-specific value to re-enable OS-supported class 0 interrupts.

e. Write the `INT_Mask_class1` register with an OS-specific value to re-enable OS-supported class 0 interrupts.

f. Write the `INT_Mask_class2` register with an OS-specific value to re-enable OS-supported class 0 interrupts.

50. Issue a DMA command to save the low 16 KB of LS in the CSA:

    a. Write x‘0000_0000’ to `MFC_LSA` register.

    b. Write the CSA effective-address bits[0:32] to the `MFC_EAH` register.

    c. Write the CSA effective-address bits[33:63] to the `MFC_EAL` register.

    d. Write x’4000’ (transfer size of 16 KB) to `MFC_Size` register.

    e. Write x‘00’ to `MFC_TagID` register.

    f. Write '0020' to the `MFC_ClassID_Cmd` register (**put** command, with `TClassID` and `RClassID` of '00').

    g. Read the `MFC_CmdStatus[Rc]` register field and, if value is not equal to ‘00’, go back to Step 50a on page 363.

51. Write ‘0’ to the `SPU_NPC[IE]` register field, and write the `SPU_NPC[LSA]` register field to the LS entry-point address of the SPU Context-Save Sequence code[26].

52. Write the `SPU_Sig_Notify_1` register with the upper 32 bits of the effective address in the CSA where the copy of the LSCSA is stored. (This register is read in Step 6 on page 364.)

53. Write the `SPU_Sig_Notify_2` register with the lower 32 bits of the effective address in the CSA where the copy of the LSCSA is stored. (This register is read in Step 8 on page 364.)

54. Issue a DMA command to the copy the SPU Context-Save Sequence code from main-storage to the low 16 KB of the LS and start the SPU:

    a. Write the LS beginning address of the SPU Context-Save Sequence code to the `MFC_LSA` register.

    b. Write the most-significant word of the main-storage address of the SPU Context-Save Sequence code to the `MFC_EAH` register.

    c. Write the least-significant word of the main-storage address of the SPU Context-Save Sequence code to the `MFC_EAL` register.

    d. Write the SPU-Context-Save Sequence code size to the `MFC_Size` register.

    e. Write x‘00’ to `MFC_TagID` register.

    f. Write '004A' to the `MFC_ClassID_Cmd` register (**getfs** command, with `TClassID` and `RClassID` of '00').

    g. Read the `MFC_CmdStatus[Rc]` register field and, if value is not equal to ‘00’, go back to Step 54a on page 363.

55. Write the `Prxy_QueryMask` register to ‘1’ (enable Tag-Group 0). Then, issue a PPE **eieio** instruction.

56. Poll the `Prxy_TagStatus[g0]` register field until it reads ‘01’ (Tag-Group 0 complete), or write `Prxy_QueryType[TS]` to ‘01’ and wait for a Tag-Group Complete interrupt. Then, write the `INT_Stat_class2[T]` register field until it reads ‘1’.

---

26. See *Section 12.5.1.2* on page 364 for this sequence.

57. Poll the `SPU_Status[R]` register field until it reads ‘0’, or wait for an SPU Class 0 or Class 2 interrupt. Then, reset interrupt conditions by writing the `INT_Stat_class0` register with ‘1’ for each interrupt handled, and by writing the INT_Stat_class2`[T]` bit with ‘1’.
58. If the `SPU_Status[P]` register field is set to ‘1’ and `SPU_Status[StopCode]` reads Success, the context-save has succeeded. Otherwise, the context-save failed.
59. Proceed to Step 26 of the PPE Context-Restore Sequence on page 367.

12.5.1.2 *SPU Context-Save Sequence*

The SPU code that performs the following context-save sequence is copied by the PPE into a portion of the low 16 KB of the LS and run in Step 54 on page 363. This sequence copies the remainder of the LS and the contents of the local-store context save area (LSCSA) into the CSA. The LSCSA consists of the GPRs, status registers, and the remaining channel state.

The SPU Context-Save sequence is:

1. Save the low 16 GPRs, using quadword stores, in the LSCSA. They are saved at this time for performance reasons.
2. Read the `SPU_RdEventMask` channel and save it in the LSCSA.
3. Read the `MFC_RdTagMask` channel and save it in the LSCSA.
4. Set the `SPU_WrEventMask` channel to ‘0’ to mask all events.
5. Set the `MFC_WrTagMask` channel to ‘01’ to unmask only Tag-Group 0.
6. Read the `SPU_RdSigNotify1` channel data to obtain the upper 32-bits of the CSA effective address.
7. Write the high address from Step 6 to the `MFC_EAH` channel.
8. Read the `SPU_RdSigNotify2` channel data to obtain the low 32-bits of the CSA effective address.
9. Update the low effective addresses in a pre-canned, 15-element (16 KB each) DMA list that saves the most-significant 240 KB[27] of the LS to the CSA.
10. Enqueue a **putl** (Tag-Group 0) command for the DMA list updated in Step 9, page 364.
11. Save the high 112 GPRs, using quadword stores, in the LSCSA.
12. Issue the floating-point status and control register read instruction (**fscrrsd**) and save the status in the LSCSA.
13. Read and save the `SPU_RdDec` channel data in the LSCSA.
14. Read and save the `SPU_RdSRR0` channel data in the LSCSA.
15. Enqueue a **putllc** command, using an effective address in the CSA, to remove any possible lock-line reservation.
16. Enqueue a **put** (Tag-Group 0) command to save the LSCSA to the CSA.
17. Enqueue an **mfcsync** (Tag-Group 0) command to the `MFC_Cmd` register.
18. Write the `MFC_WrTagUpdate` channel with ‘01’ (update tag status on any completion).

27. The size may differ from 240 KB, depending on the value of the SPU_LSLR register.

19. Read the `MFC_RdTagStat` channel data. This stalls the SPU until DMA Tag-Group 0 is complete.
20. Read the `MFC_RdAtomicStat` channel data. This stalls until the **putllc** command completes.
21. Issue a stop-and-signal (**stop**) instruction with the status set to a Success status. If there is a software-detected error in the SPU save-context code, the stop-and-signal status should be set to a Diagnostic status instead of a Success status.

12.5.2 Context-Restore Sequence

12.5.2.1 *PPE Context-Restore Sequence for SPE*

This section outlines how to fully restore the context of an SPE.

If the SPE context was successfully saved, as described in *Section 12.5.1.1* on page 359, proceed to Step 26 on page 367. Otherwise, harvest an SPE by performing the following steps:

1. If multithreading is being used, acquire an SPU mutual-exclusion kernel lock.
2. Disable Interrupts from this SPE:
   - Write the `INT_Mask_class0` register with ‘0’ (disable all class 0 interrupts).
   - Write the `INT_Mask_class1` register with ‘0’ (disable all class 1 interrupts).
   - Write the `INT_Mask_class2` register with ‘0’ (disable all class 2 interrupts).
3. Inhibit problem-space access to this SPE, if provided, by unmapping the virtual pages assigned to the SPE MMIO problem space.
4. If required, notify the using application that the SPE task has been terminated.
5. Set software Context Switch Pending flag.
6. Remove other SPEs’ access to this SPE by unmapping this SPE’s pages from their address spaces.
7. Write the `MFC_CNTL[Dh, Sc, Sm]` register fields to ‘1’, ‘1’, ‘0’, respectively, to suspend the queue and halt the decrementer.
8. Poll the `MFC_CNTL[Ss]` register field until ‘11’ is returned (queue suspended).
9. If the `SPU_Status[R]` register field is set to ‘1’ (running):
   - If `SPU_Status[E]` is set to ‘1’, poll `SPU_Status[R]` until it reads ‘0’ (stopped).
   - If `SPU_Status[L]` is set to ‘1’, or if `SPU_Status[IS]` is set to ‘1’:
     a. Write the `SPU_RunCntl[Run]` register field to ‘00’ (stop request).
     b. issue a PPE **eieio** instruction.
     c. Poll `SPU_Status[R]` until it reads ‘0’ (stopped).
     d. Write `SPU_RunCntl[Run]` to ‘10’ (isolate exit request).
     e. Issue a PPE **eieio** instruction
     f. Poll `SPU_Status[R]` until it reads ‘0’ (stopped).
   - If `SPU_Status[W]` is set to ‘1’:
     a. Write `SPU_RunCntl[Run]` to ‘00’ (stop request).

   b. Issue a PPE **eieio** instruction.
   c. Poll `SPU_Status[R]` until it reads ‘0’ (stopped).
- Go to Step 11 on page 366.

10. If `SPU_Status[R]` is set to ‘0’ (stopped):
    - If `SPU_Status[E]` is set to ‘1’:
      a. Write `MFC_SR1[S]` to ‘1’.
      b. Write `SPU_RunCntl[Run]` to ‘01’(run request).
      c. Issue a PPE **eieio** instruction.
      d. Poll `SPU_Status[R]` until it reads ‘0’ (stopped).
    - If `SPU_Status[L]` is set to ‘1’, or if `SPU_Status[IS]` is set to ‘1’:
      a. Write `MFC_SR1[S]` to ‘1’.
      b. Write `SPU_RunCntl[Run]` to ‘10’ (exit request).
      c. Issue a PPE **eieio** instruction.
      d. Poll `SPU_Status[R]` until it reads ‘0’ (stopped).
    - If there is any other value, no action is needed; the SPU is stopped in a non-isolated state.
11. Write the `MFC_MSSync` register. Then, poll `MFC_MSSync[P]` for a value of ‘0’.
12. Write all of the `MFC_TLB_Invalidate_Entry[IS,VPN,L,Lp]` register fields to ‘0’. Then, issue a PPE **sync** instruction.
13. Handle any pending interrupts from this SPE. This is OS- or hypervisor-specific. One option is to re-enable interrupts to handle any pending interrupts, with the interrupt handlers recognizing the software Context Switch Pending flag, to ensure that the SPE execution or MFC command queue is not resumed.
14. Write the `MFC_CNTL[Pc]` register field to ‘1’ (purge queue).
15. Poll `MFC_CNTL[Ps]` until it reads ‘11’ (purge complete).
16. Write the `SPU_PrivCntl[S,Le,A]` register fields to ‘0’ (reset).
17. Set the `SPU_LSLR` register to the size of implemented LS, minus one.
18. Write the `MFC_SR1[D]` bit to ‘0’, the `MFC_SR1[S]` bit to ‘1’, and the `MFC_SR1[TL,R,PR, and T]` bits set correctly for the OS environment.
19. If the `MFC_SR1[R]` bit is set to ‘1’, write ‘0’ to the `SLB_Invalidate_All` register.
20. For the following channels:
    - `SPU_RdEventStat` (channel index 0).
    - `SPU_WrEventMask (channel index 1)`.
    - `SPU_RdSigNotify1 (channel index 3)`.
    - `SPU_RdSigNotify2 (channel index 4)`.
    - `MFC_RdTagStat (channel index 24)`.
    - `MFC_RdListStallStat (channel index 25)`.
    - `MFC_RdAtomicStat (channel index 27)`.
      a. Write the channel index to the `SPU_ChnlIndex` register. Then, issue an **eieio** instruction.

   b. Write ‘0’ to the `SPU_ChnlData` register.

   c. Write ‘0’ to the `SPU_ChnlCnt` register[28].

21. For the following channels:
    - `MFC_Cmd` (channel index 21, count = 16).
    - `MFC_WrTagUpdate (channel index 23`, count = 1).
    - `SPU_WrOutMbox` (channel count 28, count = 1).
    - `SPU_RdInMbox` (channel count 29, count = 0).
    - `SPU_WrOutIntrMbox` (channel count 30, count = 1).

    a. Write the channel index to the `SPU_ChnlIndex` register. Then, issue a PPE **eieio** instruction.

    b. Write the count to the `SPU_ChnlCnt` register.

22. If the OS is using interrupts instead of polling:

    a. Write the `INT_Stat_class0` register with -1 (reset all class 0 interrupts).

    b. Write the `INT_Stat_class1` register with -1 (reset all class 1 interrupts).

    c. Write the `INT_Stat_class2` register with -1 (reset all class 2 interrupts).

    d. Write the `INT_Mask_class0` register with OS-specific value (reset OS-supported class 0 interrupts).

    e. Write the `INT_Mask_class1` register with OS-specific value (reset OS-supported class 1 interrupts).

    f. Write the `INT_Mask_class2` register with OS-specific value (reset OS-supported class 2 interrupts).

23. Set the software Context Switch Active flag.

24. Write the `MFC_TClassID` register with the value 0x10000000 (`TClassID0` Quota=16).

25. Write the `MFC_CNTL[Sc]` register field to ‘0’ and `MFC_CNTL[Sm]` to ‘0’ (resumes queue processing).

At this point, either the SPE’s context has been saved in the CSA or an SPE has been harvested. A context can now be restored from the CSA to the SPE, as described in the following steps:

26. Set a software watchdog timer to the maximum time for the restore sequence.

27. If any of the `SPU_Status[I,S,H, P]` bits are set to ‘1’ in the CSA, then add the correct instruction sequence to the end of the SPU Context-Restore Sequence code[29], after the context-restore stop-and-signal (Step 22 on page 373) to restore the following values to the `SPU_Status` register:
    - `SPU_Status[P]` set to ‘1’: Stop-and-signal instruction, followed by a branch instruction to itself[30].
    - `SPU_Status[I]` set to ‘1’: Illegal instruction, followed by a branch instruction to itself.
    - `SPU_Status[H]` set to ‘1’: Unconditional halt instruction[31] followed by a branch instruction to itself.

28. The channel count need not be restored for the SPU_WrEventMask channel.
29. See *Section 12.5.2.2* on page 372.
30. A branch instruction to itself can be encoded in assembly as “br.”.

- SPU_Status[S] set to ‘1’: Two no-op instructions, followed by a branch instruction to itself.
- SPU_Status[S, I] set to ‘1’: Illegal instruction, followed by a branch instruction to itself.
- SPU_Status[S, P] set to ‘1’: Stop-and-Signal instruction with the stop code from the saved SPU_Status, followed by a branch instruction to itself.
- SPU_Status[P, H] set to ‘1’: Unconditional halt instruction, followed by a Stop-and-Signal instruction with the saved stop code from the saved SPU_Status, followed by a branch instruction to itself.
- SPU_Status[P, I] set to ‘1’: Illegal instruction, followed by a Stop-and-Signal instruction with the saved stop code from the SPU_Status, followed by a branch instruction to itself.

28. If all the SPU_Status bits[I, S, H, P, R] bits are set to ‘0’ in the CSA (SPU was not executing), then add a branch instruction to itself to the end of the SPU Context-Restore Sequence code, after the context-restore stop-and-signal (Step 22 on page 373).
29. Restore the RA_Group_ID and RA_Enable registers from the CSA.
30. If the MFC_SR1[R] register field is set to ‘1’:
    a. Write ‘0’ to the SLB_Invalidate_All register.
    b. Write x‘0’ to SLB_Index register.
    c. Write the SLB_VSID register with the VSID to the SPU Context-Restore code.
    d. Write the SLB_ESID register with the ESID to the SPU Context-Restore code.
    e. Write ‘1’ to SLB_Index register.
    f. Write the SLB_VSID register to provide access to the CSA.
    g. Write the SLB_ESID register to complete providing access to the CSA.
31. Write the SPU_NPC[IE] register field to ‘0’ and SPU_NPC[LSA] to the LS entry-point address of the SPU Context-Restore code.
32. Write the SPU_Sig_Notify_1 register with the upper 32 bits of the CSA effective address.
33. Write the SPU_Sig_Notify_2 register with the lower 32 bits of the CSA effective address.
34. If the MFC_CNTL[Ds] is set to ‘1’ in the CSA (decrementer was running), adjust the decrementer value in the LSCSA copy by the amount of time that elapsed between the context-save and the context-restore, as follows:
    a. Read the Time Base (TB) register value.
    b. Subtract the value from the time-base value saved in the CSA during the save sequence.
    c. Subtract that value from Decrementer value in the LSCSA copy.
    d. Save the result in the LSCSA copy.
    e. Write the software Decrementer Status Running flag in the LSCSA copy.
    f. If the updated Decrementer value in the LSCSA copy is negative, then set the software Decrementer Wrapped flag in the CSA.
35. Copy the SPU_Out_Mbox data from the CSA into the CSA’s copy of the LSCSA.
36. Copy the SPU_Out_Intr_Mbox data from the CSA into the CSA’s copy of the LSCSA.

---

31. A unconditional halt instruction can be encoded in assembly as “heq r#,r#”, where # is any register.

37. Issue a DMA command to copy the SPU Context-Restore Sequence code to a portion of the low 16 KB of the LS, and start the SPU:
    a. Write the beginning LS address of the SPU Context-Restore Sequence to the `MFC_LSA` register.
    b. Write the beginning CSA effective-address bits[0:32] of the SPU Context-Restore Sequence code to the `MFC_EAH` register.
    c. Write the beginning CSA effective-address bits[33:63] of the SPU Context-Restore Sequence code to the `MFC_EAL` register.
    d. Write the size of the SPU Context-Restore Sequence code to the `MFC_Size` register.
    e. Write x'00' to the `MFC_TagID` register.
    f. Write '0048' to the `MFC_ClassID_Cmd` register (**gets** command, with `TClassID` and `RClassID` of '00').[32]
    g. Read the `MFC_CmdStatus[Rc]` register field and, if the value is not '00', go back to Step 37a on page 369.
38. Write the `Prxy_QueryMask` register to '1' (enable Tag-Group 0), and issue a PPE **eieio** instruction.
39. Poll the `Prxy_TagStatus[g0]` register field until it reads '01' (Tag-Group 0 complete), or write `Prxy_QueryType[TS]` with '01' and wait for a Tag-Group Complete interrupt. Then, write the `INT_Stat_class2[T]` register field with '1'.
40. Poll the `SPU_Status[R]` register field until it reads '0', or wait for an SPU Class 0 or Class 2 interrupt. Then, reset Interrupt conditions by writing the `INT_Stat_class0` register with '1' for each interrupt handled, and by writing the INT_Stat_class2`[T]` bit with '1'.
41. If the `SPU_Status[P]` register field is set to '1' and `SPU_Status[StopCode]` reads Restore Successful, the context-restore succeeded. Otherwise, the context-restore failed, and the sequence should be exited.
42. Restore the `SPU_PrivCntl` register from the CSA.
43. If any of the `SPU_Status[I, S, H, P]` bits are set to '1' in the CSA, restore the error or single-step state, as follows:
    a. Write the `SPU_RunCntl[Run]` register field to '01' (Run request).
    b. Issue a PPE **eieio** instruction. Then, poll `SPU_Status[R]` until it is set to '0'.
    c. Go to Step 45 on page 369.
44. If all of the `SPU_Status[I, S, H, P, R]` bits are set to '0' in the CSA, restore as follows:
    a. Write `SPU_RunCntl[Run]` to '01' (Run request).
    b. Issue a PPE **eieio** instruction. Then, poll `SPU_Status[R]` until it reads '1' (running).
    c. Write `SPU_RunCntl[Run]` to '00' (Stop request).
    d. Issue a PPE **eieio** instruction. Then, poll `SPU_Status[R]` until it reads '0' (stopped).
45. Issue a DMA command to restore the low 16 KB of LS from the CSA:
    a. Write x'0000_0000' to `MFC_LSA` register.

32. See *Section 12.5.2.2* on page 372 for the sequence to be copied and run.

   b. Write the effective-address bits[0:32] for the beginning of the CSA's copy of the LS to the `MFC_EAH` register
   c. Write the effective-address bits [33:63] for the beginning of the CSA's copy of the LS to the `MFC_EAL` register.
   d. Write the value 16 KB to `MFC_Size` register.
   e. Write x'00' to `MFC_TagID` register.
   f. Write '0040' to the `MFC_ClassID_Cmd` register (**get** command, with `TClassID` and `RClassID` of '00').
   g. Read the `MFC_CmdStatus[Rc]` register field. If the value is not equal to '00', go back to Step 45a on page 369.

46. Poll the `Prxy_TagStatus[g0]` register field until it reads '01' (Tag-Group 0 complete), or write the `Prxy_QueryType[TS]` register field to '01' and wait for Tag-Group Complete interrupt. Then, write the `INT_Stat_class2[T]` register field to '1'.
47. Write the `MFC_CNTL[Sc]` register field to '1' and the `MFC_CNTL[Sm]` to '0'.
48. Poll `MFC_CNTL[Ss]` until it reads '11' (queue suspended).
49. Write all of the `MFC_TLB_Invalidate_Entry[IS,VPN,L, Lp]` register fields to '0's. Then, issue a PPE **sync** instruction.
50. If the OS is using interrupts instead of polling:
    a. Write the `INT_Mask_class0` register with '0' (disable all class 0 interrupts).
    b. Write the `INT_Mask_class1` register with '0' (disable all class 1 interrupts).
    c. Write the `INT_Mask_class2` register with '0' (disable all class 2 interrupts).
    d. Write the `INT_Stat_class0` register with -1 (reset all class 0 interrupts).
    e. Write the `INT_Stat_class1` register with -1 (reset all class 1 interrupts).
    f. Write the `INT_Stat_class2` register with -1 (reset all class 2 interrupts).
51. If the `MFC_CNTL[Q]` bit is set to '0' in the CSA (MFC command queues were not empty), then restore 96 doublewords to the `MFC_CQ_SR` registers from the CSA.
52. Restore the `Prxy_QueryMask` register from the CSA.
53. Restore the `Prxy_QueryType` register from the CSA.
54. Restore the `MFC_CSR_TSQ` register from the CSA.
55. Restore the `MFC_CSR_Cmd1` and `MFC_CSR_Cmd2` registers from the CSA.
56. Restore the `MFC_CSR_ATO` register from the CSA.
57. Restore the `MFC_TClassID` register from the CSA.
58. Set the Lock-Line Reservation Lost Event:
    a. If `SPU_Channel_0_Count` is set to '0' in the CSA, and `SPU_WrEventMask[Lr]` is set to '1 in the CSA', and `SPU_RdEventStat[Lr]` is set to '0' in the CSA, then set `SPU_RdEventStat_Count` to '1' in the CSA.
    b. Set the `SPU_RdEventStat[Lr]` to '1' in the CSA in order to set Lock-Line Reservation Lost event.
59. If the status of the CSA software Decrementer Wrapped flag is set:

a. If the `SPU_Channel_O_Count` is set to ‘0’ in the CSA, and the `SPU_WrEventMask[Tm]` is set to ‘1’ in the CSA, and the `SPU_RdEventStat [Tm]` is set to ‘0’ in the CSA, then set the `SPU_RdEventStat_Count` to ‘1’ in the CSA.

b. Set the `SPU_RdEventStat[Tm]` to ‘1’ in the CSA in order to set a Decrementer event.

60. For the following channels:
    - `SPU_RdEventStat` (channel index 0).
    - `SPU_RdSigNotify1` (channel index 3).
    - `SPU_RdSigNotify2` (channel index 4).
    - `MFC_RdTagStat` (channel index 24).
    - `MFC_RdListStallStat` (channel index 25).
    - `MFC_RdAtomicStat` (channel index 27).

    a. Write the channel index to the `SPU_ChnlIndex` register, and issue a PPE **eieio** instruction.

    b. Restore channel data to the `SPU_ChnlData` register from the CSA.

    c. Restore channel count to the `SPU_ChnlCnt` register from the CSA.

61. For the following channels:
    - `MFC_WrMSSyncReq` (channel index 9, count = 1).
    - `MFC_Cmd`, (channel index 21, count from CSA).
    - `MFC_WrTagUpdate`, (channel index 23, count = 1).

    a. Write the channel index to the `SPU_ChnlIndex` register, and issue a PPE **eieio** instruction.

    b. Restore the channel count to the `SPU_ChnlCnt` register from the count.

62. Restore the `SPU_LSLR` register from the CSA.
63. Restore the `SPU_Cfg` register from the CSA.
64. Restore the `PM_Trace_Tag_Wait_Mask` register from the CSA.
65. Restore the `SPU_NPC` register from the CSA.
66. For the `SPU_RdInMbox` channel:

    a. Write the channel index value 29 to the `SPU_ChnlIndex` register, and issue a PPE **eieio** instruction.

    b. Restore the `SPU_ChnlCnt` register from the `SPU_RdInMbox_Count` in the CSA.

    c. Restore the `SPU_ChnlData` register from the CSA’s `SPU_RdInMbox` data0 area.[33]

    d. Restore the `SPU_ChnlData` register from the CSA’s `SPU_RdInMbox` data1 area.

    e. Restore the `SPU_ChnlData` register from the CSA’s `SPU_RdInMbox` data2 area.

    f. Restore the `SPU_ChnlData` register from the CSA’s `SPU_RdInMbox` data3 area.

67. If `SPU_Mbox_Stat[P]` is set to ‘0’ in the CSA (mailbox empty), read from the `SPU_Out_Mbox` register (discarding the data).

---

33. This was saved in the CSA in Step 43c on page 362.

68. If `SPU_Mbox_Stat[I]` is set to ‘0’ in the CSA (interrupt mailbox empty):
    a. Read from the `SPU_Out_Intr_Mbox` register (discarding the data).
    b. Write the `INT_Stat_Class2[M]` register field to ‘1’ (reset Mailbox interrupt status).
69. If `MFC_SR1[R]` is set to ‘1’ in the CSA:
    a. Restore the `SLB_Index` register from the CSA.
    b. For index=0 through index=7:
        1. Write the index value to the `SLB_Index` register, and issue a PPE **eieio** instruction.
        2. Restore the `SLB_ESID` register from `SLB_ESID[index]` in the CSA.
        3. Restore the `SLB_VSID` register from `SLB_VSID[index]` in the CSA.
    c. Restore the `SLB_Index` register from the CSA.
70. Restore the `MFC_SR1` register from the CSA.
71. Restore other SPE mappings to the SPE.
72. If `SPU_Status[R]` is set to ‘1’ in the CSA (running), write the `SPU_RunCntl[Run]` register field to ‘01’.
73. Restore the `MFC_CNTL` register from the CSA.
74. Restore mapping of the PPE-side problem-state access to the SPE resources.
75. Reset the Context Switch Active flag.
76. Re-enable SPE interrupts:
    - Restore the `INT_Mask_class0` register from the CSA.
    - Restore the `INT_Mask_class1` register from the CSA.
    - Restore the `INT_Mask_class2` register from the CSA.
77. If multithreading is being used, release software SPU mutual-exclusion lock.

12.5.2.2 *SPU Context-Restore Sequence*

The following context-restore sequence is copied by the PPE into a portion of the low 16 KB of the LS and run in Step 37f on page 369:

1. Set the `SPU_WrEventMask` channel to ‘0’ to mask all events.
2. Set the `MFC_WrTagMask` channel to ‘01’ to unmask only Tag-Group 0.
3. Read the `SPU_RdSigNotify1` channel data to obtain the high address of the CSA effective address.
4. Write the high address from Step 3 to the `MFC_EAH` channel.
5. Read the `SPU_RdSigNotify2` channel data to obtain the low address of the CSA effective address.
6. Update the low effective addresses in a pre-canned, 15-element (16 KB each) DMA list that restores the most-significant 240 KB of the LS from the CSA.
7. Enqueue a **get** (Tag-Group 0) command to restore the LSCSA copy from the CSA to the LSCSA (in low LS memory).
8. Enqueue a **getl** (Tag-Group 0) command for the DMA list updated in Step 6, page 372.

9. Write the `MFC_WrTagUpdate` channel with a value of ‘01’ to update the tag status on any completion.
10. Read the `MFC_RdTagStat` channel data. This stalls the SPU until the DMA Tag-Group 0 instruction is complete.
11. Issue quadword loads for the high 112 GPRs from the LSCSA.
12. If the LSCSA software Decrementer Running flag is set, then write the `SPU_WrDec` channel with the decrementer value from the LSCSA.
13. Enqueue a **putllc** command using an effective address in the CSA, to remove any possible lock-line reservation.
14. Read the `MFC_RdAtomicStat` channel data. This stalls the SPU until the **putllc** completes.
15. Write the `SPU_WrOutMbox` channel with the `SPU_WrOutMbox` data from the LSCSA.
16. Write the `SPU_WrOutIntrMbox` channel with the `SPU_WrOutIntrMbox` data from the LSCSA.
17. Restore the Floating-Point Status and Control Register (FPSCR) from the LSCA using the **fscrwr** instruction.
18. Restore the Save and Restore Register 0 (`SRR0`) data from the LSCSA and write to the `SPU_WrSSR0` channel.
19. Read the `SPU_RdEventMask` data from the LSCSA and write to the `SPU_WrEventMask` channel.
20. Read the `MFC_RdTagMask` data from the LSCSA and write to the `MFC_WrTagMask` channel.
21. Restore the low 16 GPRs from the LSCSA using quadword loads.
22. Issue a stop-and-signal (**stop**) instruction with Success status.
23. There may be additional instructions placed here to restore the correct stopped state (see Steps 27 and 28 on page 367).

If there is a software-detected error in the SPU Context-Restore Sequence, the `SPU_RdInMbox` channel may be written with a Diagnostic status. A halt conditional instruction should then be issued, followed by a branch instruction to itself.

12.6 Considerations for Hypervisors

The sequences above assume an OS that is running at both Privilege 1 and Privilege 2 levels—that is, an operating environment without a separate hypervisor and without support for logical-partition switching. If a hypervisor is used and the SPE to be switched is allocated to a logical partition, additional state may need to be saved, set, and restored. This state may include some or all of the following:

- MFC Logical Partition ID Register (`MFC_LPID`).
- Interrupt Routing Register (`INT_Route`).
- MFC Storage Description Register (`MFC_SDR`).
- SMM Hardware Implementation Dependent Register (`SMM_HID`).
- MFC Real-Mode Address Boundary Register (`MFC_RMAB`).
- 256 TLB Entries.
- MFC TLB Index Register (`MFC_TLB_Index`).

- MFC TLB Real-Page Number Register (`MFC_TLB_RPN`).
- MFC TLB Virtual-Page Number Register (`MFC_TLB_VPN`).
- MFC TLB Replacement Management Table Data Register (`MFC_TLB_RMT_Data`).
- MFC command request order is lost.
- DMA and register lock state for address translation exceptions is lost.

See *Section 11 Logical Partitions and a Hypervisor* on page 327 for details about these resources.

13. Time Base and Decrementers

13.1 Introduction

The CBE Time-Base facility provides the timing functions for the CBE Core-Clock (`NClk`) domain. The facility includes two software-visible 64-bit time-base registers—one for configuration and one for counting—and eleven software-visible 32-bit decrementers—three in the PPE (two for user software, and one for Hypervisor software) and one in each of the eight SPEs (for user software).

The counting time-base (called the TB register) and decrementer registers count time in *ticks*, at a ticking rate called the *time-base frequency*, regardless of the Slow-State setting[34]. The TB register and all decrementers are updated at the same time-base frequency. The TB register provides a long-period counter; it holds a monotonically increasing value, and it is split into upper and lower halves. the 32-bit decrementers support shorter-period counting and most of them provide interrupt-signaling after a specified amount of time has elapsed.

The TB register and the four PPE decrementers are implemented in the CBE processor as PPE Special-Purpose Registers (SPRs). All of these registers are volatile and must be initialized during startup. The eight SPE decrementers are implemented in the CBE processor as SPU channel registers that are accessed by SPE software using channel instructions (**`rdch`** or **`wrch`**).

13.2 Time Base Facility

13.2.1 Clock Domains

The CBE processor has three clock domains, each running asynchronously to the other two domains:

- *CBE Core Clock (`NClk`)*—This clock times the PPU, SPUs, and parts of the PPSS and MFCs. The CBE Core Clock (`NClk`) is occasionally referred to as the Core Clock (`CORE_CLK`) or Core Clock Frequency (CCF).
- *MIC Clock (`MiClk`)*—This clock times the Memory Interface Controller (MIC).
- *BIC Core Clock (`BClk`)*—This clock times the Bus Interface Controller (BIC), which is part of the Cell Broadband Engine Interface (BEI) Unit to the I/O interface.

The following CBE logic blocks run as half the CBE Core Clock frequency (`NClk/2`):

- Element Interconnect Bus (EIB) and interfaces to the EIB (parts of the PPSS and MFCs).
- I/O Interface Controller (IOC).
- MIC logic that is part of the CBE Core.
- BIC logic that is part of the CBE Core.
- Pervasive logic, which is the logic that provides power management, thermal management, clock control, software-performance monitoring, trace analysis, and so forth.

34. See *Section 15.1.1 CBE Slow State* on page 424 for information about the Slow State.

The time base and decrementer facilities described in this chapter fall entirely within the CBE Core Clock (`NClk`) domain.

13.2.2 Time-Base Registers

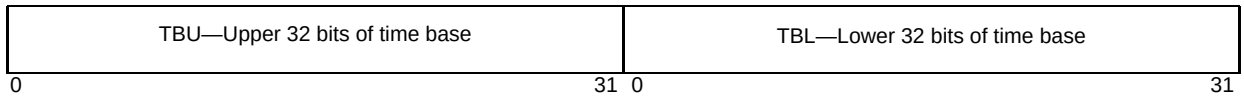
There are two registers with the name “Time Base Register” but they have different acronyms:

- *Time Base Register (TB)*—This SPR register contains the time-base count value, as described in *Section 13.2.3* on page 377. The physical implementation of this register actually includes the following SPR registers:
  - Time Base Register, Read Only (TB), SPR #268.
  - Time Base Upper Register, Read Only (TBU), SPR #269.
  - Time Base Lower Register, Write Only (TBL), SPR #284.
  - Time Base Upper Register, Write Only (TBU), SPR #285.
- *Time Base Register (TBR)*—This MIMO register specifies the time-base sync mode, as described in *Section 13.2.4* on page 378, and the internal reference-clock divider setting, as described in *Section 13.2.3* on page 377.

For details on the TBR register, see the *Cell Broadband Engine Registers* specification.

The TB register, shown in *Figure 13-1*, is a 64-bit structure that contains a 64-bit unsigned integer. The register is divided into two halves, a 32-bit TB Upper (TBU) register and a 32-bit TB Lower (TBL) register. Each tick pulse to the TB register adds ’1’ to the low-order bit, bit[31], of TBL.

Figure 13-1. Time Base (TB) Register



The TB register can be read by user or supervisor software but written only by hypervisor software. The register must be software-enabled to allow updates. Set SPR HID6[tb_enable] to 1 to enable the time-base facility. Software reads and writes of the TB register are always allowed, regardless of SPR HID6[tb_enable]. Writes take precedence over updates (increments). Updates are ignored if they collide with a write operation.

The TB register increments until its value becomes 0xFFFFFFFF_FFFFFFFF ($2^{64} – 1$). At the next increment its value becomes 0x00000000_00000000. There is no exception or explicit indication when this occurs. If the update frequency of the TB register is constant, the register can be used as a source of values that increase at a constant rate, such as for time stamps in trace entries. Even if the update frequency is not constant, values read from the TB register are monotonically increasing (except when the value in the TB register wraps from $2^{64} – 1$ to 0). If a trace entry is recorded each time the update frequency changes, the sequence of TB-register values can be postprocessed to become actual time values.

13.2.3 Time-Base Frequency

Each tick pulse to the TB register causes it to increment by one, if it is software-enabled to allow updates. There are two modes for updating the TB register:

- *Internal Time-Base Sync Mode:* In this mode, a Phase Locked Loop (PLL) Reference Clock (`PLL_REFCLK`) external input signal and a Core Clock Multiplier (`CCM`) are used to synchronize and increment the TB register. See *Section 13.2.4.1* on page 378 for details.
- *External Time-Base Sync Mode:* In this mode, the rising edge of the Time-Base Enable (TBEN) external input signal is used to synchronize and increment the TB register. See *Section 13.2.4.3* on page 381 for details.

In the Internal Time-Base Sync Mode, the CBE Core Clock (`NClk`) frequency is:

```
NClk = PLL_REFCLK * CCM
```

The maximum time-base frequency is constrained by the following factors:

- PLL Reference Clock (`PLL_REFCLK`) frequency.
- Core Clock Multiplier (`CCM`), also called the PLL Multiplier Setting (`PLLmultiplier`).
- Power-Management CBE Slow State (called *Slow Mode* in parameters described in this chapter). For details, see *Section 15.1.1 CBE Slow State* on page 424.

The equations for maximum time-base frequency (`Fmax`) are:

```
RefDiv >= ⌈MaxSlowModeNclkDivider * 11 / CCM⌉

Fmax (Internal Mode) = PLL_REFCLK / RefDiv

Fmax (External Mode) <= PLL_REFCLK / (1 + (11 * MaxSlowModeNclkDivider / CCM))
```

The term `MaxSlowModeNClkDivider` denotes the maximum CBE Core Clock (`NClk`) frequency divider in CBE Slow State configured by the operating system. The CBE Power Management hardware supports eight different CBE Slow-State dividers (also called slow-mode dividers): 0 through 7. The `MaxSlowModeNClkDivider` value for each of these settings are 1, 2, 3, 4, 5, 6, 8, and 10 respectively. The value is set in the Power Management Control Register (`PMCR`). See `PMSR[BE_Slow_n]` for the current CBE Slow State setting, and see *Section 15.1.1 CBE Slow State* on page 424 for information on how to change power management states.

The standard 3.2 Ghz CBE Core Clock (`NClk`) is configured by specifying the Internal Time-Base Sync State, a 400 MHz PLL Reference Clock (`PLL_REFCLK`) frequency, and a Core Clock Multiplier (`CCM`) of x8.

If the operating system uses all the CBE Slow State settings—that is to say, the `MaxSlowModeN-ClkDivider` is 10—the maximum time-base frequency is 28.57 MHz in Internal Time-Base Sync Mode and 27.11 MHz in External Time-Base Sync Mode. The time-base frequency of such a system is limited to 200 MHz with CBE Slow State disabled and in Internal Time-Base Sync Mode.

The equations above account for the worst-case CBE Slow State setting, which guarantees that the time-base frequency remains constant during CBE Slow State and during transitions between CBE Slow States.

13.2.4 Time-Base Sync Mode Controls

The SPR HID6[tb_enable] bit enables the time-base facility. After enabling, the time-base sync mode (internal or external) can be selected in the Time Base MMIO Register (TBR) using the `TBR[Timebase_mode]` bit. In the Internal Time-Base Sync Mode, the TB register should be initialized and `TBR[Timebase_setting]` is used to specify the desired time-base frequency, up to the maximum, `Fmax`.

13.2.4.1 *Internal Time-Base Sync Mode*

In Internal Time-Base Sync Mode, `TBR[Timebase_setting]` is set according to the maximum time-base calculation. The value in `TBR[Timebase_setting]` can vary from 0 to 255. If it is ‘0’, the Internal Time-Base Sync Mode is disabled and the TB register is not counting. If it is any other value, the `TBR[Timebase_setting]` value is used to determine the `RefDiv` value, as described in *Section 13.2.4.2* on page 381.

Given our previous example of a 400 MHz PPL reference clock (`PLL_REFCLK`) and x8 Core Clock Multiplier, in which all Slow State settings are supported, then:

```
RefDiv >= ceil(10 * 11 / 8) = 14.
Fmax = 400 Mhz / 14 = 28.57 MHz.
```

The maximum time-base frequency of 28.57 MHz is configured by setting `TBR[Timebase_setting]` to 0x’94’. This value is determined by finding the Timebase_setting entry in *Table 13-1* on page 379 equal to `RefDiv` of 14. If a slower time-base is desired—for example 5 MHz (a `RefDiv` value of 80)—then the `TBR[Timebase_setting]` register field would be set to 0x’A7’.

To enable Internal Time-Base Sync Mode operation, follow these steps:

1. Initialize the TB register.
2. Set `TBR[timebase_mode]` to ‘1’, and `TBR[Timebase_setting]` to ‘0’.
3. Set `TBR[timebase_mode]` to ‘0’, and `TBR[Timebase_setting]` to the value corresponding to the desired `RefDiv` value (see *Table 13-1*).

A timing diagram of steps 2 and 3 of this sequence is shown in *Figure 13-2*. In a multiple-CBE-processor system, TBEN can be used to synchronize the start of updates to TB registers across all CBE processors. As shown in *Figure 13-2*, the time base of each CBE processor does not start until after TBEN becomes ‘1’. In a single-CBE-processor system, TBEN can be tied high and the TB register starts updating after Step 2.

Figure 13-2. Internal Time-Base Sync Mode Initialization Sequence

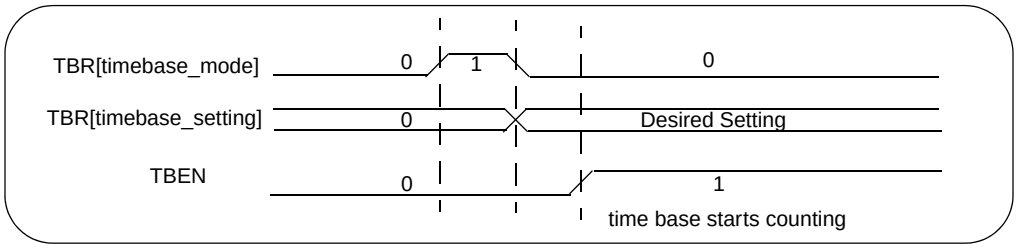


Table 13-1. TBR[Timebase_setting] for Reference Divider (RefDiv) Setting (Page 1 of 3)

RefDiv	Timebase_setting	RefDiv	Timebase_setting	RefDiv	Timebase_setting	RefDiv	Timebase_setting
0[1]	00	64	7B	128	86	192	37
1	FF	65	3D	129	43	193	9B
2	7F	66	9E	130	21	194	4D
3	3F	67	CF	131	10	195	26
4	9F	68	67	132	88	196	93
5	4F	69	B3	133	44	197	49
6	27	70	D9	134	A2	198	24
7	13	71	EC	135	D1	199	92
8	09	72	76	136	E8	200	C9
9	84	73	BB	137	74	201	64
10	42	74	DD	138	BA	202	32
11	A1	75	EE	139	5D	203	19
12	50	76	77	140	AE	204	0C
13	28	77	3B	141	D7	205	06
14	94	78	9D	142	EB	206	03
15	CA	79	4E	143	F5	207	81
16	E5	80	A7	144	7A	208	C0
17	F2	81	53	145	BD	209	E0
18	F9	82	A9	146	DE	210	70
19	7C	83	54	147	6F	211	B8
20	BE	84	2A	148	B7	212	5C
21	5F	85	95	149	DB	213	2E
22	AF	86	4A	150	ED	214	97
23	57	87	A5	151	F6	215	4B
24	AB	88	52	152	FB	216	25

1. This setting disables Internal Time-Base Sync Mode.

Table 13-1. TBR[Timebase_setting] for Reference Divider (RefDiv) Setting (Page 2 of 3)

RefDiv	Timebase_setting	RefDiv	Timebase_setting	RefDiv	Timebase_setting	RefDiv	Timebase_setting
25	55	89	29	153	7D	217	12
26	AA	90	14	154	3E	218	89
27	D5	91	8A	155	1F	219	C4
28	EA	92	45	156	0F	220	E2
29	75	93	22	157	87	221	71
30	3A	94	91	158	C3	222	38
31	1D	95	48	159	61	223	1C
32	0E	96	A4	160	B0	224	8E
33	07	97	D2	161	58	225	47
34	83	98	69	162	2C	226	23
35	C1	99	B4	163	96	227	11
36	60	100	5A	164	CB	228	08
37	30	101	2D	165	65	229	04
38	18	102	16	166	B2	230	02
39	8C	103	8B	167	59	231	01
40	46	104	C5	168	AC	232	80
41	A3	105	62	169	D6	233	40
42	51	106	31	170	6B	234	A0
43	A8	107	98	171	B5	235	D0
44	D4	108	CC	172	DA	236	68
45	6A	109	E6	173	6D	237	34
46	35	110	73	174	B6	238	1A
47	9A	111	39	175	5B	239	8D
48	CD	112	9C	176	AD	240	C6
49	66	113	CE	177	56	241	E3
50	33	114	E7	178	2B	242	F1
51	99	115	F3	179	15	243	78
52	4C	116	79	180	0A	244	BC
53	A6	117	3C	181	05	245	5E
54	D3	118	1E	182	82	246	2F
55	E9	119	8F	183	41	247	17
56	F4	120	C7	184	20	248	0B
57	FA	121	63	185	90	249	85
58	FD	122	B1	186	C8	250	C2
59	7E	123	D8	187	E4	251	E1

1. This setting disables Internal Time-Base Sync Mode.

Table 13-1. TBR[Timebase_setting] for Reference Divider (RefDiv) Setting (Page 3 of 3)

RefDiv	Timebase_setting	RefDiv	Timebase_setting	RefDiv	Timebase_setting	RefDiv	Timebase_setting
60	BF	124	6C	188	72	252	F0
61	DF	125	36	189	B9	253	F8
62	EF	126	1B	190	DC	254	FC
63	F7	127	0D	191	6E	255	FE

1. This setting disables Internal Time-Base Sync Mode.

13.2.4.2 *Determining the System Time-Base Frequency*

Assuming the time base is configured as Internal Time-Base Sync mode, then the frequency can be determined using the following procedure:

1. Verify that the system is configured in Internal Time-Base Sync mode by reading the TBR register and verifying that the `TBR[Timebase_mode]` field is '0'.
2. Determine the `RefDiv` value by doing a inverse table lookup of *Table 13-1* using the value stored in `TBR[Timebase_setting]`. For example, if the `TBR[Timebase_setting]` value is 0x'2A', then the `RefDiv` is 84. If `TBR[Timebase_setting]` is 0, then the time base is disabled.
3. Compute the time-base frequency by multiplying the `PLL_REFCLK` and `RefDiv`.

The number of cycles, either PPE or SPE cycles, per time-base tick is further reduced based upon the current Slow State setting, `PMSR[BE_Slow_n]`.

```
cycles per time-base tick = PLL_REFCLK * RefDiv / SlowModeFactor.
```

where the SlowModeFactor is either 1, 2, 3, 4, 5, 6, 8, or 10 corresponding to the PMSR[BE_Slow_n] values of 0 through 7.

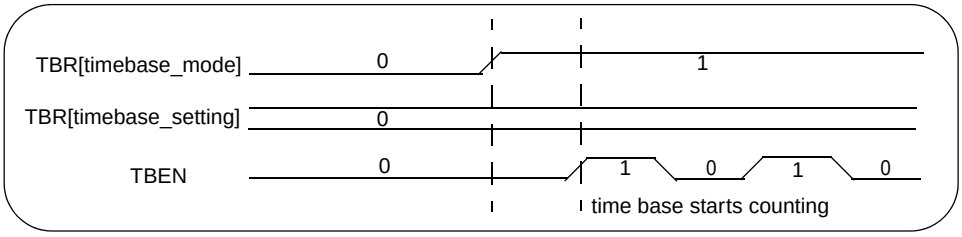
Since the time-base registers are accessible only by privileged software, the operating environment should provide information on the time-base frequency. Consult system documentation for specific details.

13.2.4.3 *External Time-Base Sync Mode*

To select the External Time-Base Sync Mode, set `TBR[timebase_mode]` to '1'. The value of `TBR[Timebase_setting]` has no effect, although the no effect, but the recommended value is x'00'.

In the External Time-Base Sync Mode, the rising edge of the TBEN signal is used to generate the TB-register ticks. Similar to Internal Time-Base Sync Mode, TBEN can be used to synchronize the start of the time base across different CBE processors in a multi-CBE-processor system. A timing diagram is shown in *Figure 13-3* on page 382. In a single-CBE-processor system, TBEN can be active before `TBR[timebase_mode]` is set to '1'.

Figure 13-3. External Time-Base Sync Mode Initialization Sequence



13.2.5 Reading and Writing the TB Register

13.2.5.1 *Reading the TB Register*

Either user or supervisor software can read the contents of the TB register into a GPR. To read the contents of the TB register into register **r**A, execute the following instruction:

```
mftb    rA
```

The above example uses the simplified (extended) mnemonic form of the **mftb** instruction (equivalent to **mftb r**A,**268**). Using this instruction copies the entire TB register (TBU || TBL) into **r**A. Reading the TB register has no effect on the value it contains or the periodic incrementing of that value.

If the simplified mnemonic form **mftbu r**A (equivalent to **mftb r**A**,269**) is used, the contents of TBU are copied to the low-order 32 bits of **r**A, and the high-order 32 bits of **r**A are cleared (0 || TBU).

13.2.5.2 *Writing the TB Register*

Only hypervisor software can write the contents of a GPR into the TB register. The simplified (extended) mnemonics, **mttbl** and **mttbu**, write the lower and upper halves of the TB register, respectively. The simplified mnemonics are for the **mtspr** instruction. The **mtspr**, **mttbl**, and **mttbu** instructions treat TBL and TBU as separate 32-bit registers; setting one leaves the other unchanged. It is not possible to write the entire 64-bit TB register in a single instruction.

The instructions for writing the TB register are not dependent on the implementation or mode. Thus, code written to set the TB register on a 32-bit implementation will work correctly on the CBE processor.

The TB register can be written by a sequence such as:

```
lwz     rx,upper                #load 64-bit value for
lwz     ry,lower                # TB into rx and ry
li      rz,0
mttbl   rz                      #force TBL to 0
mttbu   rx                      #set TBU
mttbl   ry                      #set TBL
```

Provided that no exceptions occur while the last three instructions are being executed, loading 0 into TBL prevents the possibility of a carry from TBL to TBU while the TB register is being initialized.

13.2.6 Computing Time-of-Day

Since the update frequency of the TB register is system-dependent, the algorithm for converting the current value in the TB register to time-of-day is also system-dependent.

In a system in which the update frequency of the TB register may change over time, it is not possible to convert an isolated time-base value into time of day. Instead, a time-base value has meaning only with respect to the current update frequency and the time of day at which the update frequency was last changed.

Each time the update frequency changes, either the system software is notified of the change by means of a Decrementer interrupt, or the change was instigated by the system software itself. At each such change, the system software must compute the current time of day using the old update frequency, compute a new value of ticks-per-second for the new frequency, and save the time of day, time-base value, and tick rate. Subsequent calls to compute time of day use the current time-base value and the saved data.

A generalized service to compute time-of-day could take the following as input:

- Time-of-day at beginning of current epoch.
- Time-base value at beginning of current epoch.
- Time-base update frequency.
- Time-base value for which time-of-day is desired.

For a CBE system in which the time-base update frequency does not vary, the first three inputs would be constant.

13.3 Decrementers

The CBE processor contains eleven software-visible decrementers, three in the PPE and one in each SPE. All decrementers tick at the same frequency as the same time-base frequency, subject to control commands which turn them on and off.

13.3.1 PPE Decrementers

The PPE has three 32-bit software-visible decrementers:

- Two DEC Decrementers, one per PPE thread, accessible to supervisor software.
- One HDEC Hypervisor Decrementer, accessible to hypervisor software.

Each update tick to the PPE decrementers causes their values to decrease by one, at the time-base frequency. SPR `HID6[tb_enable]` must be software-enabled to allow PPE decrementer updates.

The DEC Decrementers support short-term counting and provide a means of signaling an interrupt after a specified amount of time has elapsed, unless the Decrementer is altered by software in the interim, or the time-base frequency changes.

The HDEC Hypervisor Decrementer provides a means for hypervisor software to manage timing functions independently of the Decrementers, which are managed by user or supervisor software running in virtual partitions. Similar to the Decrementer, the HDEC is a counter that is updated at the time-base frequency, and it provides a means of signaling an interrupt after a specified amount of time has elapsed. For more about HDEC, see *Section 11 Logical Partitions and a Hypervisor* on page 327.

13.3.1.1 *Decrementer Operation*

The DEC counts down, causing a Decrementer interrupt (unless masked by `MSR[EE]`) when it passes through ‘0’. The DEC satisfies the following requirements:

- The operations of the TB and DEC registers are driven by the same time-base frequency.
- Loading a GPR from the DEC has no effect on the DEC.
- Storing the contents of a GPR to the DEC replaces the value in the DEC with the value in the GPR.
- Whenever bit[0] of the DEC is set to ‘1’, a Decrementer interrupt is signaled. Multiple Decrementer interrupts may be received before the first interrupt occurs; however, any additional requests are canceled when the interrupt occurs for the first request.
- If the DEC is altered by software and the content of bit [0] is set to ‘1’, a Decrementer interrupt is signaled.

In systems that change the time-base frequency for purposes such as power management, the Decrementer input frequency will also change. Software must be aware of this in order to set interval timers.

13.3.1.2 *Writing and Reading the Decrementers*

The contents of the DEC can be read or written using the **mfspr** and **mtspr** instructions, both of which are supervisor instructions when they refer to the DEC. Using a simplified (extended) mnemonic for the **mtspr** instruction, the DEC may be written from GPR **r**A with the following:

```
mtdec   rA
```

Using a simplified (extended) mnemonic for the **mfspr** instruction, the DEC may be read into GPR **r**A with the following:

```
mfdec   rA
```

13.3.2 SPE Decrementers

13.3.2.1 *Decrementer Operation*

Each SPU contains one 32-bit decrementer implemented as a down-counter. All decrementers in the SPUs and the PPE count down at the same rate. An SPU decrementer is accessed through two channels—the SPU Write Decrementer channel (`SPU_WrDec`) and the SPU Read Decrementer channel (`SPU_RdDec`)—as described in *Section 17.4 SPU Decrementer* on page 447.

An SPU decrementer can be run, read, and halted by SPU channel instructions, which can also enable or disable decrementer events. A decrementer event becomes pending when the value in the decrementer changes from ‘0’ to negative (the most-significant bit changes from ‘0’ to ‘1’). See *Section 18.6.9 Procedure for Handling the SPU Decrementer Event* on page 480 for details on event-interrupt handling.

13.3.2.2 *Writing and Reading the Decrementers*

The SPU decrementer is accessed by reading and writing SPU channels using the **rdch** and **wrch** instructions or using the **spu_readch** and **spu_writech** intrinsics. The decrementer value is read from the SPU Read Decrementer channel, and the SPU Write Decrementer channel allows the decrementer value to be set. In addition, the SPU Write Event Mask channel and the SPU Write Event Acknowledgement channel are used to turn the decrementer on and off, and to handle decrementer events.

For more information on the SPU decrementer channels, see *Section 17 SPE Channel and Related MMIO Interface* on page 441.

13.3.3 Using an SPU Decrementer to Monitor SPU Code Performance

The following example demonstrates the use of an SPU decrementer to monitor code performance. In the example, profiling macros are inserted into the source code around the sections of code to be timed. The resulting timing information is output to the console, but it could instead be read and processed by an application program.

13.3.3.1 *Sample Performance-Profiling Code*

The following sample code illustrates a general approach to performance profiling of the SPEs. The sample uses a profile-checkpoint facility which allows timing of code sections. Source code to be timed can be instrumented with macros, such as the `prof_cp(N)` macros shown here (where N is a number) that resolve to special no-op instructions which the profiling program interprets as commands to output instruction- and cycle-count information. The `prof_clear()`, `prof_start()`, and `prof_stop()` macros have additional meaning: `prof_clear()` clears the performance information, `prof_start()` starts recording of performance data, and `prof_stop()` turns off recording of the data.

The following shows the use of the profiling macros:

```
#include "profile.h"

...
prof_clear();
prof_start();
...
<code to be timed>
...
prof_stop();
```

Additional `prof_cp(N)` macros can be inserted in the timed section for finer-grained timing information.

13.3.3.2 *Hardware Code Performance*

Because the granularity of the decrementer-based profiling implementation is limited by the frequency of the SPU decrementer, the code sections to be timed should typically execute several thousand SPU cycles in order to obtain reasonable accuracy. The implementation is not suitable for extremely fine-grained timing of SPU code.

13.3.3.3 *SPU Decrementer Profiling*

In the code sample below, the total number of elapsed decrementer ticks is kept in a global variable called `_count` (each SPU has its own copy). Another global variable, `_count_base`, keeps track of the last value read from the decrementer. To allow timing to be turned on and off, a boolean variable called `_counting` keeps track of whether or not profiling is enabled. These variables are defined as:

```
extern volatile unsigned long long _count

extern volatile unsigned int _count_base

extern volatile _Bool _counting;
```

In the body of the `prof_cp()` macro, if profiling is enabled, the decrementer value is read from the SPU Read Decrementer channel, using the `spu_readch` intrinsic, and subtracted from the `_count_base`. This gives the elapsed number of decrementer ticks since the last profiling-macro invocation (the decrementer counts down with each tick). If the decrementer value is greater than the `_count_base`, an underflow condition has occurred; underflow can also occur if end is less than `_count_base`, but we ignore that situation for the moment.

Assuming a single decrementer underflow, the elapsed number of ticks is given by `0x'FFFFFFFF' + _count_base - end`. The global `_count` is incremented by the elapsed ticks calculated above, and a message displaying the count is printed to the console output. Finally, in order to avoid counting the time spent in the profiling macro itself, the decrementer is read once again and assigned to the `_count_base` variable.

The following code sample implements the routine described above:

```
#define DECR_MAX   0xFFFFFFFF
#define DECR_COUNT DECR_MAX

#define prof_cp(cp_num)                                        \
{                                                              \
   if (_counting) {                                            \
     unsigned int end = spu_readch(SPU_RdDec);                 \
     _count += (end > _count_base) ?                           \
       (DECR_MAX + _count_base - end) :                        \
       (_count_base - end);                                    \
   }                                                           \
   printf("SPU#: CP"#cp_num", %llu\n", _count);                \
   _count_base = spu_readch(SPU_RdDec);                        \
}
```

The `prof_clear()` macro is implemented similarly, except that it also clears the timing information after displaying it:

```
#define prof_clear()                                          \
{                                                             \
   if (_counting) {                                           \
     unsigned int end = spu_readch(SPU_RdDec);                \
     _count += (end > _count_base) ?                          \
       (DECR_MAX + _count_base - end) :                       \
       (_count_base - end);                                   \
   }                                                          \
   printf("SPU#: clear, %llu\n", _count);                     \
   _count = 0;                                                \
    _count_base = spu_readch(SPU_RdDec);                      \
}
```

The `prof_start()` macro performs the additional work of enabling the decrementer. This is done by using an `spu_writech` intrinsic to write a new decrementer count to the SPU Write Decrementer channel, and another `spu_writech` to the SPU Write Event Mask channel to enable decrementer events:

```
#define prof_start()                                          \
{                                                             \
   if (_counting) {                                           \
     unsigned int end = spu_readch(SPU_RdDec);                \
     _count += (end > _count_base) ?                          \
       (DECR_MAX + _count_base - end) :                       \
       (_count_base - end);                                   \
   }                                                          \
   printf("SPU#: start, %llu\n", _count);                     \
   spu_writech(SPU_WrDec, DECR_COUNT);                        \
   spu_writech(SPU_WrEventMask, MFC_DECREMENTER_EVENT);       \
   _counting = 1;                                             \
    _count_base = spu_readch(SPU_RdDec);                      \
}
```

Finally, the `prof_stop()` macro disables counting by writing the SPU Write Event Mask channel to disable decrementer events, and writing the SPU Write Event Acknowledgement channel to acknowledge pending events and stop the decrementer, again using the `spu_writech` intrinsic:

```
#define prof_stop()                                           \
{                                                             \
   if (_counting) {                                           \
     unsigned int end = spu_readch(SPU_RdDec);                \
     _count += (end > _count_base) ?                          \
       (DECR_MAX + _count_base - end) :                       \
       (_count_base - end);                                   \
   }                                                          \
   printf("SPU#: stop, %llu\n", _count);                      \
```

```
   _counting = 0;                                        \
   spu_writech(SPU_WrEventMask, 0);                      \
   spu_writech(SPU_WrEventAck, MFC_DECREMENTER_EVENT);   \
   _count_base = spu_readch(SPU_RdDec);                  \
}
```

Commonality can be exploited by defining shared macros for the global `_count` and `_count_base` updates that appear in all of the profiling macros and in the interrupt handler.

13.3.3.4 *Decrementer Underflow Handling*

If the code sections to be profiled are known to execute in less than 2^{32} decrementer ticks, the simple implementation shown above is sufficient for providing timing information. For longer-running code sections, the profiling implementation needs to handle cases in which the decrementer underflows multiple times between profiling-macro invocations. This can be done by installing an interrupt handler that executes whenever the decrementer reaches 0, which triggers an SPU decrementer event.

In the body of the handler, the handler checks for the SPU decrementer event and increments the global `_count` as above. It then handles the event by clearing the SPU decrementer-event bit in the SPU Write Event Mask channel using the `spu_writech` intrinsic. An `spu_writech` to the SPU Write Event Acknowledgement channel acknowledges the decrementer event and also causes the decrementer to stop. The decrementer is then restarted by writing a new value to the SPU Write Decrementer Count channel and setting the SPU decrementer event bit in the SPU Write Event Mask channel.

The code for the second-level interrupt handler is:

```
static unsigned int spu_decr_profile_slih(unsigned int events)
{
 if (events & MFC_DECREMENTER_EVENT) {
   if (_counting) {
     unsigned int end = spu_readch(SPU_RdDec);
     _count += (end > _count_base) ?
       (DECR_MAX + _count_base - end) :
       (_count_base - end);
   }
    events &= ~MFC_DECREMENTER_EVENT;
   spu_writech(SPU_WrEventMask, events);
   spu_writech(SPU_WrEventAck, MFC_DECREMENTER_EVENT);
   spu_writech(SPU_WrDec, DECR_COUNT);
   spu_writech(SPU_WrEventMask, MFC_DECREMENTER_EVENT);
   _count_base = spu_readch(SPU_RdDec);
 }
 return (events);
}
```

The handler should be installed as an event-specific handler that is dispatched from a common event handler, as described in *Section 18.6.9 Procedure for Handling the SPU Decrementer Event* on page 480. In addition, SPU events must be enabled when profiling is turned on (and can be disabled when profiling is turned off). Enabling and disabling of interrupts is done using the `spu_ienable` and `spu_idisable` intrinsics. The initialization of the handler and enabling of interrupts can be inserted into the body of the `prof_start()` macro (with logic to ensure that the handler only gets initialized once), and disabling of interrupts can be added to the `prof_stop()` macro.

14. Objects, Executables, and SPE Loading

This chapter describes processor-specific aspects of object-file formats that are used for program linking and loading. Topics discussed include the PPE- and SPE-specific extensions to the standard Executable and Linking Format (ELF) object-file formats, runtime loading of programs into an SPE, and the CBEA Embedded SPE Object Format (CESOF).

CESOF allows SPE executable object files to be embedded inside PPE object files. It supports a mechanism for linking SPE and PPE object files that access shared system memory objects through the globally defined symbols. This chapter also describes basic operating-system protocols for loading and executing programs from a CESOF object file.

14.1 Introduction

The CBE processor is a heterogeneous multiprocessor not only because the SPEs and the PPE have different architectures but also because they have disjoint address spaces and different models of memory and resource protection. The PPE can run a virtual-memory operating system, so it can manage and access all system resources and capabilities. In contrast, the SPUs are not intended to run an operating system, and SPE programs can access the main-storage address space, called the effective-address (EA) space, only indirectly through the DMA controller in their MFC. The two processor architectures are different enough to require two distinct tool chains for software development.

The tool chains for both the PPE and SPE processor elements produce object files in the ELF format. ELF is a flexible, portable container for relocatable, executable, and shared object (dynamically linkable) output files of assemblers and linkers; this format is described in *Section 14.2* on page 392. We refer to PPE-ELF and SPE-ELF to differentiate between ELF for the two architectures. CESOF is an application of PPE-ELF that allows PPE executable objects to contain SPE executables; this format is described in *Section 14.5* on page 402.

To ease the development of combined PPE-SPE multiprocessor programs, the CBE operating-system model uses CESOF and provides SPE process-management primitives. Though programmers may often keep in mind a heterogeneous model of the CBE processor when dividing an application program into concurrent threads, the CESOF format and, for example, the Linux™ operating-system thread APIs allow programmers to focus on application algorithms instead of managing basic tasks such as SPE process creation and global variable sharing between SPE and PPE threads.

The following sections describe the ELF format for PPE and SPE object and executable files, low-level support and requirements for loading SPE programs, the CESOF format for combining PPE and SPE object files, and the operating-system model of loading and running concurrent, cooperative SPE and PPE programs.

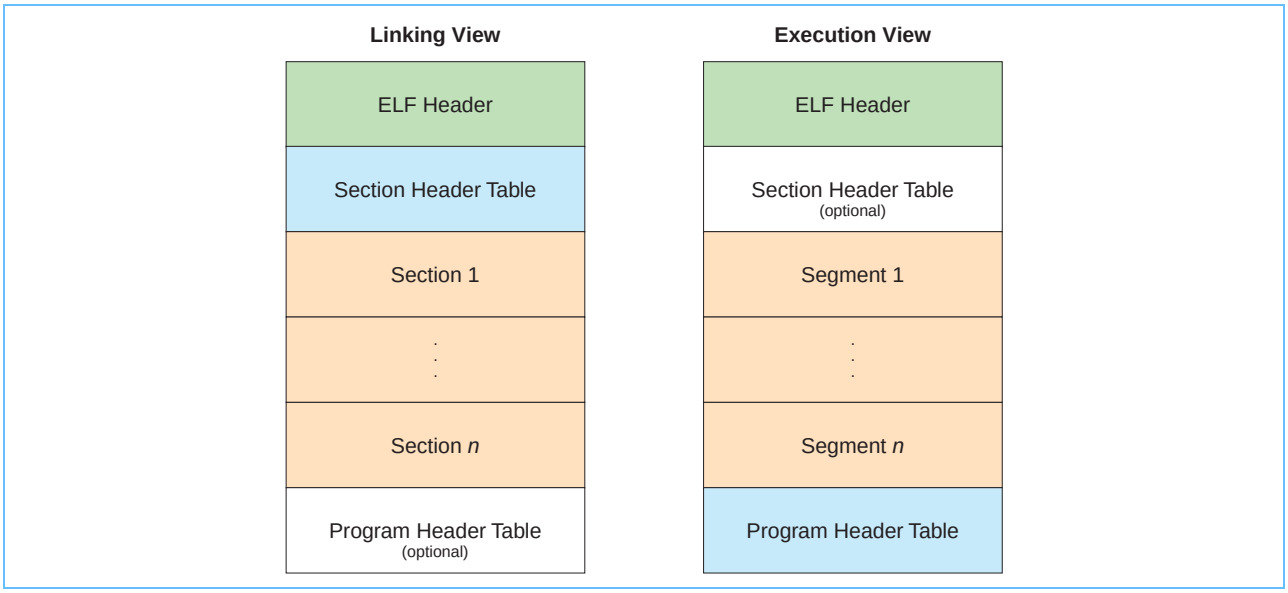
14.2 ELF Overview and Extensions

14.2.1 Overview

ELF object files are binary representations of programs intended to execute directly on CBE processor elements. To organize the program information in an object file for efficient processing by linkers and loaders, ELF defines a simple hierarchy of structures. Object files have an ELF header that is used to find a section header or a program header; an object file may have both section and program headers. The section header specifies sections in the file for use in linking; the program header specifies program segments in the file for use in program loading. Much of an object file's information, such as processor instructions and initialized data, is contained in its section and segment data structures, but an object file also contains other data structures such as symbol tables.

The section header and the program header provide two different, parallel views of an object file's contents that reflect the different needs of linking and execution. ELF object files can be used by a linker to build an executable program object (linking view) and by an operating system or loader to build a process image in memory (execution view). *Figure 14-1* shows an object file's organization under the two views.

Figure 14-1. Object-File Format



The binary ABI specifications for the CBE processor elements, listed below, extend the basic ELF definition. The extensions result in two ELF definitions specific to the CBE processor: PPE-ELF and SPE-ELF.

For a complete discussion of the basic ELF definition and the PPE and SPE APIs, see the following documents:

- *Tool Interface Standard (TIS) Executable and Linking Format (ELF)* specification.
- *Linux Standard Base Core Specification for PPC 3.0* (http://www.linuxbase.org/spec).
- *64-bit PowerPC ELF Application Binary Interface Supplement 1.7.1*.

- *SYSTEM V APPLICATION BINARY INTERFACE PowerPC Processor Supplement* (this document specifies the 32-bit ABI).
- *Cell Broadband Engine Linux Reference Implementation Application Binary Interface Specification*.
- *SPU Application Binary Interface Specification*.

14.2.2 SPE-ELF Extensions

14.2.2.1 *ELF Header*

The machine-specific information that applies to SPE-ELF objects is shown in *Table 14-1*.

Table 14-1. SPE-ELF Header Fields

Field	Value	Comments
`e_ident[EI_CLASS]`	`ELFCLASS32`	Specifies that the CBE objects are 32-bit objects.
`e_ident[EI_DATA]`	`ELFDATA2MSB`	Specifies that the CBE objects use big-endian byte addressing.
`e_type`	`ET_NONE`	The file has no file type.
	`ET_REL`	The file is a relocatable file. The file holds code and data suitable for linking with objects to create an executable or plugin file.
	`ET_EXEC`	The file is an executable file. The file holds a fully linked program suitable for execution.
	`ET_DYN`	The file is an SPE plugin. The plugin file must contain a SPUNAME note section for each named plugin.
`e_machine`	`EM_SPU`	A value of 23 indicates that the code is to be executed on an SPE processor element.
`e_flags`	`0`	No flags have been defined; therefore, this member must contain zero.

14.2.2.2 *Symbols*

Except for the special `_EAR_` symbol-name mangling protocol used by programmers and tools to build CESOF objects (see *Section 14.5.5* on page 407), the global symbols produced by a compiler should not be *mangled*; that is, a symbol should not have any extra prefix or suffix added by the tool chain.

14.2.2.3 *Sections*

Executable SPE-ELF objects may contain the three typical sections and the special `.toe` section as shown in *Table 14-2*.

Table 14-2. SPE-ELF Special Sections

Name	Type	Attributes	Description
.bss	SPU_NOBITS	SHF_ALLOC + SHF_WRITE	This zero-length section specifies uninitialized data that contributes to the program's memory image. By definition, the program loader initializes the data with zeros.
.data	SPU_PROGBITS	SHF_ALLOC + SHF_WRITE	This section contains initialized data that contributes to the program's memory image. The data is copied from the executable file to EA space and then copied from EA space to an SPE's LS using DMA commands.
.text	SPU_PROGBITS	SHF_ALLOC + SHF_EXECINSTR	This section contains a program's executable instructions, traditionally called text. The machine code is copied from the executable file to EA space and then copied from EA space to an SPE's LS using DMA commands.
.toe	SPU_PROGBITS	SHF_ALLOC	This section contains an array of EAR (Effective-Address Reference) structures. Each EAR contains an eight-byte EA and an eight-byte area reserved for future use. All sections of the toe type are combined into a single segment in the SPE executable, and the SPE loader will overwrite the segment in SPE memory with the toe shadow data segment from the enclosing PPE executable object.

The `.toe` section is a key part of the CESOF specification and holds an array of Effective Address Entry (EAR) structures. Each EAR holds the EA of a shared object in main storage. The SPE linker gathers all the `.toe` sections together into a single loadable toe segment in the linked SPE executable. See *Section 14.5.2* and *Figure 14-6* for more information on CESOF extensions.

14.2.2.4 ***Program Header***

The program header contains the locations of the segment images within the file and other information needed to create the memory image for a program. The SPE ABI specifies two program-header notes, SPE Environment Note and SPE Name Note, as described below.

SPE Environment Note

SPE objects may contain sections of the `SHT_NOTE` with corresponding program-header elements of type `PT_NOTE` that define the attributes and runtime environment of an SPE program. *Table 14-3* provides details about the SPE environment note.

Table 14-3. SPE-ELF Environment Note

Field	Size (bytes)	Value
namesz	4	8
descsz	4	sizeof (spu_env)
type	4	1
name	8	"IBM SPE"
desc	sizeof (spu_env)	spu_env structure contents

The SPE environment note contains an instance of the `spu_env` structure in the `desc` field; entries in this structure are shown in *Table 14-4*.

Table 14-4. The `spu_env` Structure

Type	Name	Description
`Elf32_Word`	`revision`	Specifies the `spu_env` structure revision number. The initial revision number is 1. Future additions to this structure are added to the end, and the revision number is incremented.
`Elf32_Word`	`ls_size`	Specifies the size of SPE LS on which the program is targeted to run; this is the required `SPU_LSLR` (local-storage limit register) register setting. A size of zero indicates that the `SPU_LSLR` register must be set to the entire available LS address range.
`Elf32_Word`	`stack_size`	Specifies the SPE runtime stack size. This value is used for software runtime stack overflow checking.
`Elf32_Word`	`flags`	One flag is defined: `EF_SPU_ENCRYPTED` (bit 31). This flag specifies that the SPE ELF program is encrypted and must be decrypted and authenticated before being executed.

SPE Name Note

An SPE object must be identified with a lookup name string, and this name must be contained within a `SHT_NOTE` and in corresponding program-header elements of type `PT_NOTE`. The lookup name is used to identify the corresponding SPE plugin to the operating system.

Table 14-5. SPE-ELF Name Note

Field	Size (bytes)	Value
`namesz`	4	8
`descrsz`	4	The number of bytes in the `desc` field. This value must be a multiple of 4.
`type`	4	1
`name`	8	"`SPUNAME`"
`desc`	See the `descsz` field	Contains a null-terminated look-up name string that identifies the object.

The plugin name extracted from the `PT_NOTE` name note is used by a plugin management mechanism that provides services such as plugin table construction, plugin information retrieval, and loading plugin tables and plugin images to an SPE's LS.

14.3 Runtime Initializations and Requirements

The ABI documents for the CBE processor specify the initial machine state that the loaders create for program execution on both the PPE and SPE processors. The specifications spell out register initialization, stack frame initialization, and initial arguments passed from the operating system. Programming language systems use the initial machine state requirements to establish a standard environment for their application programs.

14.3.1 PPE Initial Machine State

14.3.1.1 *OS Interface for PPE Entry Point*

The C-language interface for a PPE program's entry point is conventionally declared as follows:

```
extern int main (int argc, char *argv[]; char *envp[]);
```

Table 14-6 explains the meaning of the parameters passed by the OS on behalf of the program that is invoking the application program.

Table 14-6. PPE Program Initial Parameters

Parameter	Value
`argc`	A non-negative argument count; specifies the number of arguments in the `argv` array.
`argv`	An array of argument strings; the OS ensures that `argv[argc] == 0`.
`envp`	An array of environment strings; as with `argv`, the `envp` array is terminated by a NULL pointer.

14.3.1.2 ***Register Initialization***

Table 14-7 shows the initial register values required by the ABI specifications for the CBE processor. Registers other than those listed in *Table 14-7* may be cleared when loading PPE programs, but a program that requires registers to have specific values must set them explicitly; a program must not rely on the loader to set any register other than those shown in *Table 14-7*.

Table 14-7. PPE Initial Register State

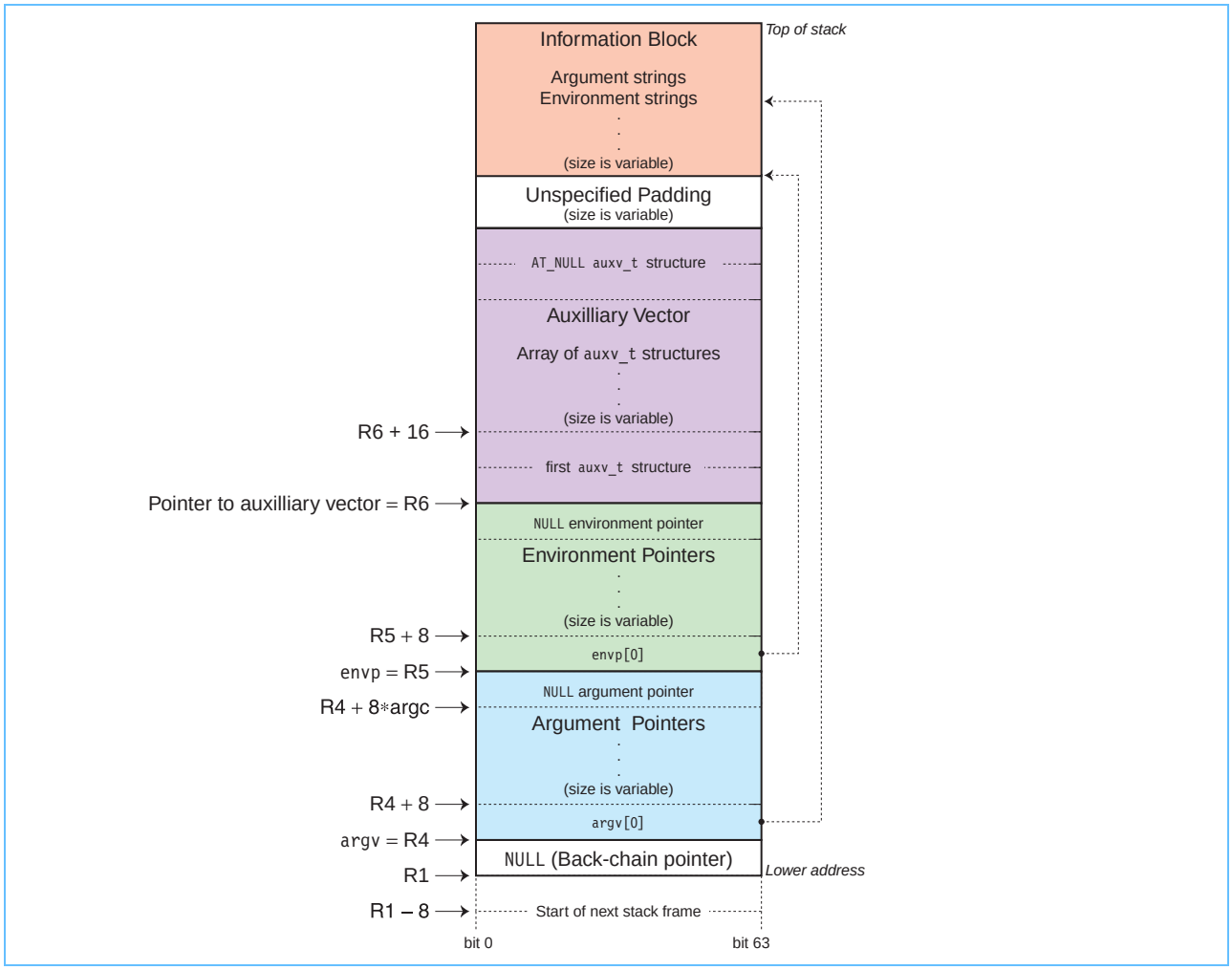
Register	Contents
R1	Initial stack pointer. The value in R1 is aligned to a quadword boundary and is a pointer to a word containing a `NULL` pointer.
R2	Initial TOC (Table Of Contents) pointer. This value is obtained through the function descriptor pointed at by the `e_entry` field in the ELF header and is used by the dynamic linker. See 64-bit PowerPC ELF Application Binary Interface Supplement 1.7 for more information.
R3	Contains `argc`, the number of arguments passed by the OS.
R4	Contains `argv`, a pointer to the array of pointers to arguments on the stack. The array is immediately followed by a `NULL` pointer; if there are no arguments, R4 points to a `NULL` pointer.
R5	Contains `envp`, a pointer to the array of pointers to environment strings on the stack. The array is immediately followed by a `NULL` pointer; if there is no environment, R5 points to a `NULL` pointer.
R6	Contains a pointer to the auxiliary vector. The auxiliary vector shall have at least one member, a terminating entry with an `a_type` of `AT_NULL`.
R7	Contains a termination-function pointer. If R7 is not zero, its value represents a function pointer that the application should register with `atexit(BA_OS)`. If R7 is zero, no action is required.
fpscr	Contains zero, which specifies “round to nearest” rounding mode, IEEE mode, and that floating-point exceptions are disabled.

14.3.1.3 ***Stack Initialization***

Before the PPE loader jumps to a PPE application’s entry point, the stack is initialized, but the CBE ABI specifications do not define a fixed stack address. A program’s stack can change from one system to another and from one process invocation to another. Thus, process initialization code must use the stack address provided in R1.

The initial stack frame, shown in *Figure 14-2*, is set up with a `NULL` back-chain pointer (at the address in R1). Above the `NULL` back-chain pointer are the three areas containing explicit and implicit arguments passed by the OS.

Figure 14-2. PPE 64-Bit Initial Stack Frame



The first area is an array of pointers to parameter strings. The second area is an array of pointers to environment strings. Both of these areas are terminated by a NULL pointer. These areas contain parameters passed from the invoking application to the invoked application, and each pointer in these areas points to a string in the information block at the top of the stack frame.

The third area is an array of auxiliary information structures; each structure is of type `auxv_t`. The definition of the structure is shown below:

```
typedef struct
{
    int a_type;
    union
    {
        long   a_val;
        void  *a_ptr;
        void (*a_fcn)();
    } a_un;
```

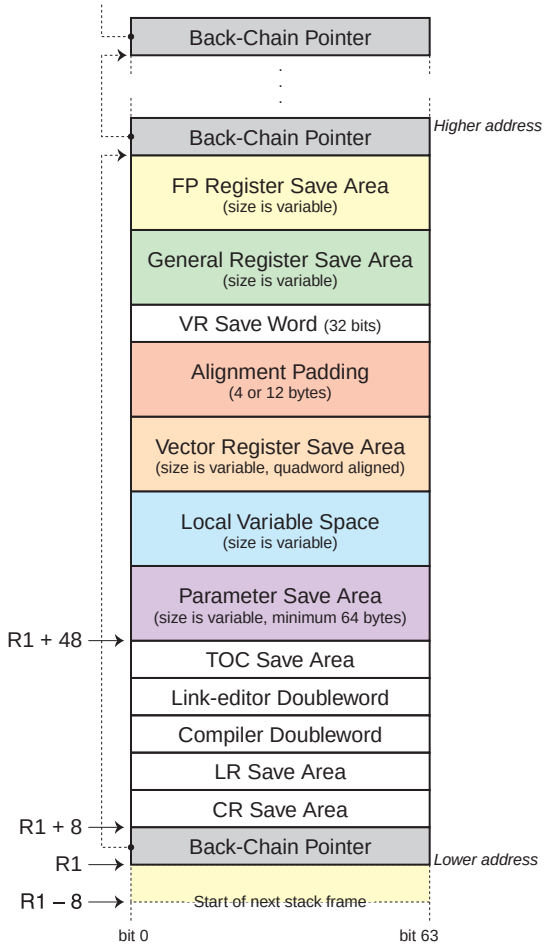
```
} auxv_t;
```

Thus, each entry in the auxiliary vector area consists of a type word and a value, pointer, or function pointer. The auxiliary vector contains information passed from the OS to the invoked application program. See the *System V Application Binary Interface PowerPC Processor Supplement* and *64-bit PowerPC ELF Application Binary Interface Supplement* documents for more information on the auxiliary vector area.

Figure 14-2 shows the layout of the PPE initial stack frame in a 64-bit execution environment. The pointers in the `argv` and `envp` arrays point into the information block. Any pointers that occur in the `auxv_t` array point into other areas of main storage.

The first application function call and subsequent calls will set up an instance of the standard SPE stack frame for a 64-bit execution environment, which is shown in *Figure 14-3*. If a calling function needs more than eight doublewords of arguments, it places the additional arguments in its parameter save area. The callee function uses an offset of 48 bytes from the back-chain pointer in its stack frame to find the additional arguments in the caller's stack frame.

Figure 14-3. PPE 64-Bit Standard Stack Frame



14.3.2 SPE Initial Machine State for Linux

This section specifies the initial SPE machine state for the Linux operating system environment. Other other operating will provide similar operating environments.

14.3.2.1 *Linux-OS Interface for SPE Entry Point*

The C-language interface for an SPE program's entry point is conventionally declared as follows in the Linux runtime environment:

```
extern int
main (unsigned long long spuid, unsigned long long argp, unsigned long long envp);
```

Table 14-8 explains the meaning of the parameters passed by the OS. This is the interface only for the Linux runtime environment. Other systems may have different interfaces. In addition, in Linux, stand alone SPU programs[35] provide a standard main interface (`int argc, char *argv[]`), where the pointers are LS pointers.

Table 14-8. SPE Program Initial Parameters

Parameter	Value
`spuid`	A unique SPU task identifier assigned by the operating system.
`argp`	An EA in main storage to application parameters.
`env`	An EA in main storage to runtime environment information.

14.3.2.2 *Register Initialization*

Table 14-9 on page 400 shows the initial register values required by the *Cell Broadband Engine Linux Reference Implementation Application Binary Interface Specification*. Registers other than those listed in *Table 14-9* may be cleared when loading SPE programs, but a program that requires registers to have specific values must set them explicitly; a program must not rely on the loader to set any register other than those shown in *Table 14-9*.

35. A programming model that allows conventional C programs to be compiled and run on an SPE, unmodified.

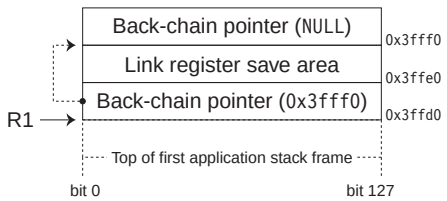
Table 14-9. SPE Register Initial Values

Register	Symbolic Name	Data Type	Comment
R1	`SP`	32-bit unsigned int	Initial stack pointer. `crt0` initializes word element 0 of R1 to point to a minimal stack frame starting at the highest LS address. For example, a system with 256-KB LS would initialize the stack pointer to `0x'3FFD0'`, and the initial, minimal stack frame would be as shown in *Figure 14-4* on page 400. The 32-bit unsigned integer in word-element 1 of the SP is used to establish the Available Stack Space. If the runtime stack size is 0, then the value of Available Stack Space is initialized to <top_of_stack> through _end (end of text/data linker symbol). Otherwise, the Available Stack Space is initialized to <stack_size>.
R3	`spuid`	64-bit unsigned int	SPU task identifier. The `spuid` value is a unique identifier assigned by the PPE OS. The `spuid` allows both the SPE program and the PPE program to identify the SPE thread (program instance) within the system.
R4	`argp`	64-bit EA	Pointer into main storage to an array of application-defined program parameters passed from the application that spawned the SPE thread.
R5	`envp`	64-bit EA	Pointer into main storage to SPU task environment structure passed from the PPE OS. The content of the structure is not defined by the CBE ABI specifications.

14.3.2.3 ***Stack Initialization***

The stack for an SPU program will be initialized with a single minimal ABI-compliant stack frame that contains three quadwords. In most system implementations, the entry function `crt0` or an equivalent will be responsible for setting up this frame. See *Figure 14-4* on page 400.

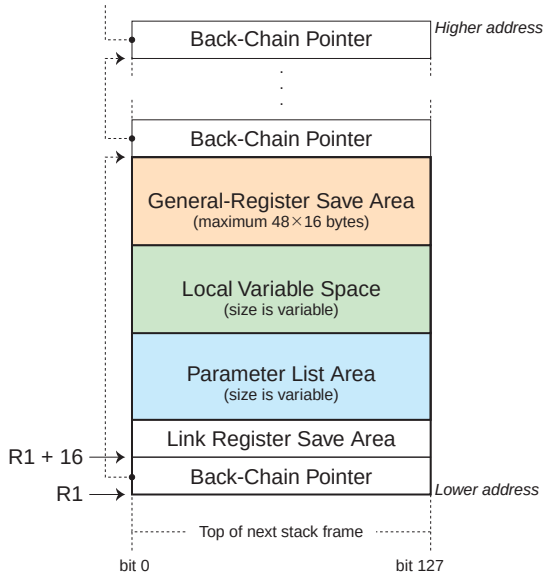
Figure 14-4. SPE Initial Stack Frame



Typically, the top of the stack is set to the highest LS quadword memory address; for a 256-KB LS, this address is `0x'3FFF0'`. The quadword at `0x'3FFF0'` contains zero, the `NULL` pointer. The quadword at `0x'3FFE0'` is the link-register save area for use by the entry function to save R0 when it calls a function. The quadword at `0x'3FFD0'` contains a back chain pointer to the top of the stack. R1 is set to `0x'3FFD0'`, which is the start of this initial stack frame.

For compliance with the ABI specification, the stack frame shown in *Figure 14-4* provides a place for the first function call in application code to save the link register. The first application function call and subsequent calls will set up an instance of the standard SPE stack frame shown in *Figure 14-5*.

Figure 14-5. SPE Standard Application Function Stack Frame



Up to 77 quadwords of arguments are passed in SPU registers in R3 through R79. If the calling function needs to pass more than 77 quadwords, it must allocate a parameter list area in its stack frame large enough to hold the additional arguments.

14.4 Linker Requirements

The extensions in the CESOF specification require some support from the PPE and SPE tool-chain link editors.

14.4.1 SPE Linker Requirements

The linker in an the SPE tool chain has some special requirements that include:

14.4.1.1 *Handling `.toe` Sections*

The SPU linker must recognize and properly handle the `.toe` section type. An SPE-ELF object need not have a `.toe` section or segment. Each `.toe` section holds the array of EAR structures (see *Section 14.5.5*), and all `.toe` sections, if any are present, must be grouped into a loadable segment in an SPE-ELF executable.

14.4.1.2 *Alignment of Loadable Segments*

The SPE linker must ensure that the loadable SPU segments in an SPU-ELF object are aligned on 16-byte boundaries and padded in length to a multiple of 16 bytes to satisfy the MFC DMA size requirement. The linker may further restrict the alignment of the segments to 128-byte

boundaries for improved transfer performance on some CBEA implementations. These alignment and size requirements allow optimal DMA performance when copying SPE-ELF loadable segments into an SPE's LS storage space.

14.4.2 PPE Linker Requirements

The special requirement for the PPE linker is the need to recognize and properly handle the sections (for example, `.spe.elf` or `.rodata.speelf`) wrapping the SPE-ELF executable image; see *Section 14.2.2*. In a PPE executable, such SPE-ELF image sections must be included by the linker in a per-system loadable segment, such as the `.text` segment.

14.5 The CESOF Format

The software-development tool chains for CBE programs conform to three standards:

- Tool Interface Standard (TIS) and Execution specification.
- Executable and Linking Format (ELF) specifications.
- PPE and SPE Application Binary Interface (ABI) specifications.

Together, these standards define an extensible, portable object-file format for relocatable and executable object files that can be statically or dynamically linked.

Because the PPE and SPE processor-element architectures are significantly different, separate tool chains are used for PPE and SPE program development. An important limitation for CBE program development is that these TIS, ELF, and ABI specifications define no way to resolve references between code and data objects for different architectures.d

To exploit the performance potential of CBE systems, CBE applications will typically contain cooperating PPE and SPE programs that share data variables. Thus, programmers need an object definition beyond the base of TIS-ELF.

The CBEA Embedded SPE Object Format (CESOF) provides three key capabilities for CBE programming environments:

- PPE and SPE programs can co-exist in a single PPE-ELF object file.
- PPE and SPE programs can share global variable references.
- The PPE link editor can accept objects that contain PPE and/or SPE code.

CESOF uses only the facilities defined in the TIS-ELF standard. Using an embedding approach requires the minimum of changes to existing PPE and SPE development tools. Since CESOF objects are PPE-ELF objects, they can participate in any program linking and execution process, static or dynamic, with other PPE-ELF objects. CESOF files may be archived, linked, and shared just like other PPE-ELF object files.

14.5.1 CESOF Overview

With CESOF, programmers can achieve some of the effects of linking PPE and SPE executables. The PPE linker can create a single PPE-ELF executable file that contains code and data for both PPE and SPE processor elements. An OS can load PPE and SPE programs that run concurrently and work cooperatively from an integrated PPE executable image. The CESOF specification includes the following mechanisms:

- The `.spe.elf` section (which can be named differently) to embed SPE-ELF executables in PPE-ELF objects
- The SPE program handle data structure, which is used by an OS runtime to access the embedded SPE program and data.
- A reserved “shadow” section to keep the EA-space bindings for the shared global data object references of the embedded SPE-ELF executable
- A runtime requirement that overlays the shadow section onto an SPE executable in LS memory before the SPE executable begins running.

Each of these mechanisms is discussed in a following section.

CESOF achieves cross-architecture linking by the following method:

- The SPE-ELF executable object defines a loadable segment that contains a copy of the Table of Effective Address References, or TOE (see *Section 14.5.5* on page 407). The TOE segment collects all the `.toe` sections from the linked SPE-ELF linkable objects.
- The CESOF-ELF (PPE-ELF) object, that includes the SPE-ELF executable image, defines another copy of the TOE, called the TOE shadow, in a `.data` section (as per-process data). The TOE shadow allocates the same space as the previous TOE segment of the embedded SPE-ELF executable.
- The PPE-ELF linker binds the references in the TOE shadow to addresses in EA space and fills the TOE shadow with the resolved (or bound) addresses.
- At runtime, the SPE loader loads the SPE-ELF image into an SPE’s LS memory and then replaces the toe segment in LS with the TOE shadow by copying the shadow over the toe segment.
- The running SPE-ELF program uses the bound addresses in its toe segment to reference shared data objects in EA main storage (using DMA commands to access the objects).

Thus, because the SPE program references shared data objects with a level of indirection through the TOE entries and the loader overwrites the toe segment with PPE-linker bound addresses, CESOF achieves the effect of the PPE linker binding PPE addresses into the SPE executable, which is not otherwise supported by the ELF specification. Thanks to the CESOF mechanisms, the linker can do this without knowing anything about the SPE architecture.

14.5.2 CESOF Use Convention of ELF

CESOF introduces new sections into PPE-ELF and SPE-ELF object respectively.

- PPE-ELF `.spe.elf` section: a container for an SPE-ELF image in a PPE-ELF object.
- SPE-ELF `.toe` section: a container for an array of EAR structures; each EAR contains an EA pointer to a shared data-object EA space. All `.toe` sections are concatenated into one toe

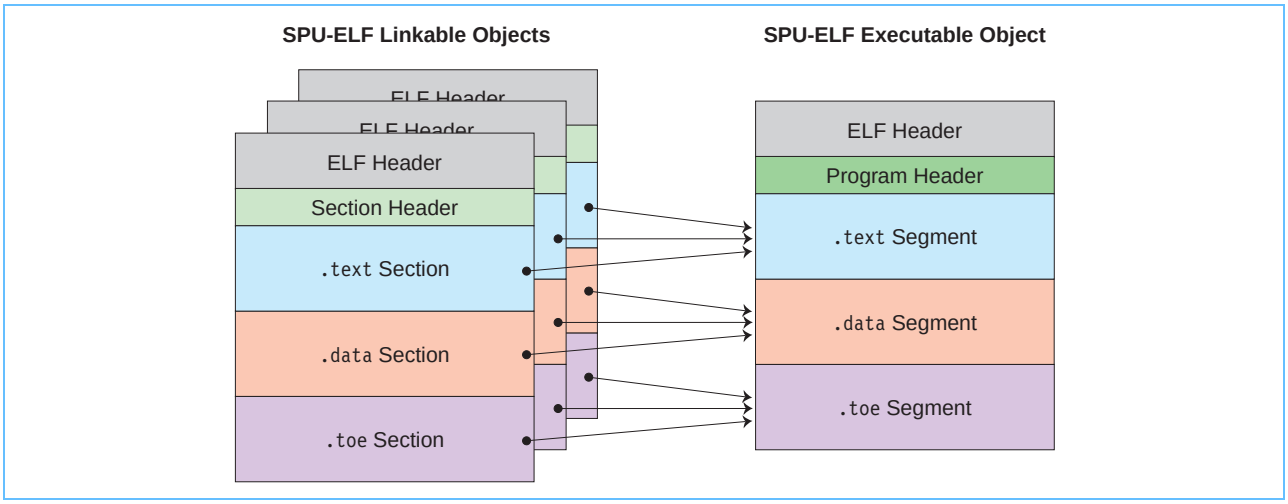
segment in an SPE-ELF executable. `.toe` sections and the segment are filled with empty EAR structures.

- PPE-ELF TOE shadow section: a copy of the above SPE-ELF toe segment is allocated in a PPE-ELF data section. This matching TOE shadow data section contains initialized EAR structures containing bound EA pointers instead of the empty EAR structures in the toe segment of the SPE executable.

When the SPE-ELF executable is loaded into an SPE's LS, the TOE shadow is copied directly over the toe segment in the LS. The copying effectively updated the EA addresses to the bound values. At run time, the SPE code executes DMA commands that use the bound addresses in the TOE in LS to access shared data objects in main storage.

Figure 14-6 shows how the SPE-ELF sections in linkable objects are combined into segments in an executable object.

Figure 14-6. Linking SPE-ELF Sections into Loadable Segments in an Executable



14.5.3 Embedding an SPE-ELF Executable in a PPE-ELF Object: The `.spu.elf` Section

An SPE program is compiled and linked by the SPE compiler and tool chain into an SPE-ELF executable file. Without the support of the mechanisms like those in CESOF, the only way a PPE program can use an SPE program is to refer to the name of the SPE-ELF executable object file. Because the PPE operating system's linker and runtime loader know nothing about SPE-ELF executables in this case, a PPE program must explicitly copy or memory-map the SPE executable image from a file to main storage and explicitly program an SPE's DMA controller to move the SPE executable to an SPE's LS.

The process of loading an SPE-ELF image from a file to an SPE's LS requires tedious programming that distracts a programmer from coding a solution to an application problem. In addition, if the SPE executable is shared by several PPE processes, storage is wasted by having a separate copy in each PPE process.

A solution to the explicit loading and the storage redundancy is to make the SPE-ELF object part of the PPE-ELF object. CESOF accomplishes this by defining a special section in PPE-ELF to include the image of the SPE-ELF executable file in the section.

The CESOF specification suggests that ".spe.elf" is used as the name for such sections. However, a toolchain may choose to use any name as long as the SPE-ELF image sections are linked and loaded into a per-system segment. The single copy of image can then be shared by more than one processes.

In assembler code, the declaration of the PPE-ELF section and file inclusion can be as shown below:

```
    .global      spu_program

    .align       7
    .section     .spu.elf, "ax", @progbits
spu_program:
    .incbin      "spu_program.elf"
```

In a PPE program, a programmer declares a symbol external to the code module that creates a reference to the SPE image:

```
extern spu_program[];
```

The result is that the SPE-ELF executable object has been embedded (or wrapped) in the `.spe.elf` section. In an executable object, all the `.spe.elf` sections may be collected in the per-system loadable segment, such as the `.text` segment, so that main storage will contain only one copy of an SPE-ELF image shared by multiple processes. The CESOF specification, however, still allows the `.spu.elf` sections to be merged into any segment, such as a data segment, depending on the needs of the hosting OS.

The PPE tool-chain linker can link the PPE program's reference to the global `spu_program` symbol defined in the `.spu.elf` section. The PPE program can use the `spu_program` symbol to access all the information in the SPE-ELF program image using the image's ELF header.

The CESOF specification requires the `.spe.elf` image in the loadable segment in the executable to be aligned to a DMA (128-byte) boundary so that the SPE-ELF image can be copied into an SPE's LS with maximum efficiency.

14.5.4 The `spe_program_handle` Data Structure

Using the simple solution shown above in *Section 14.5.3* on page 404, a problem arises when the included SPE-ELF object is linked into a shared PPE-ELF library object. If a reference is made directly from other modules in the shared library, the reference will be absolute, which causes the shared library to lose its position independence. Position independence is required to permit a shared library to be dynamically linked at a different address for each process that shares the library.

To maintain position independence in shared libraries that have embedded SPE-ELF objects, CESOF requires a level of indirection through a pointer to the embedded image. The pointer holds the absolute address that is determined by the dynamic linker when it binds the shared library into an in-memory process image. The pointer is held in an `spe_program_handle` data structure with the following definition:

```
typedef struct spe_program_handle {
    int     handle_size;                // sizeof (SPEProgramHandle)
    void*   elf_image;                  // pointer to the embedded SPE image
    void*   toe_shadow;                 // pointer to the toe overlay image
} SPEProgramHandle;
```

The `spe_program_handle` contains a `handle_size` member so software can determine whether the CESOF object was created for a 32-bit or 64-bit environment. The `elf_image` member holds the address of the embedded SPE-ELF image. The `toe_shadow` member holds the address of the TOE shadow overlay for the toe segment in the SPE-ELF image; this overlay will be copied to the location in LS memory where the toe segment would normally reside. The definition of the toe segment and the TOE shadow and their purpose is discussed below in *Section 14.5.5* on page 407.

Illustrated In assembler language, the 32-bit CESOF wrapping layer now looks as follows:

```
/* per-process section */
    .global     spe_program_handle

    /* new section with pointers for relocatability */
    .section    .data, “a”, @progbits
spe_program_handle:
    .align   2
    .int     12
    .int    _spe_elf_image
    .int    _spe_toe_shadow

    .section    .data, “a”, @progbits
   .align   7
 _spe_toe_shadow:
.extern shared_symbol_name_1
    .int0
    .int    shared_symbol_name_1
    .quad    0
.extern shared_symbol_name_2
    .int0
    .int0   shared_symbol_name_2
   .quad    0

/* per-system section */
    .section    .spe.elf, “ax”, @progbits
    .align      7
_spe_elf_image:
    .incbin     “spe_program.exe”
```

The `spe_program_handle` structure becomes part of a per-process loadable segment so that each in-memory process image has a private copy of the pointer to the SPE-ELF image. The SPE-ELF image is in the `.spe.elf` section, which becomes part of a single shared segment in physical storage through the virtual-memory mapping.

In the case that the SPE-ELF object `spe_program.exe` is wrapped into a PPE-ELF shared library object, the dynamic linker will initialize and map the PPE process image and load initial segments of PPE program text and the shared library into system storage (some or most loading may be delayed until it is demanded by the virtual-memory system). At this point, the dynamic linker can bind the reference to `_spe_elf_image` by computing an absolute (virtual) address value and storing that in the `_spe_elf_image` member of the `spu_program_handle` structure.

Because of the level of indirection through the pointer in the `spe_program_handle` structure, the PPE program code declares a reference to the SPE image as:

```
extern SPEProgramHandle spe_program_handle;
```

The example above illustrates the CESOF mechanism with assembler-language code, but the process of embedding SPE-ELF objects into PPE-ELF files can be automated. CBE application programmers can use a few simple utility programs to create PPE-ELF-compatible CESOF objects. See *Section 14.5.5.2* on page 409 and *Section 14.5.5.3* on page 409.

14.5.5 The TOE: Accessing Symbol Values Defined in EA Space

Global data objects in CBE-level programs are frequently declared and represented by PPE level variables and symbols. These data objects are shared by the interacting PPE and SPE programs. The resolved values of the PPE symbols usually represents the addresses of the global data objects. These values are useful to the sharing SPE programs for their DMA target and source.

CESOF allows the PPE symbol values to be resolved into SPE programs statically. CESOF inserts a level of indirection between the SPE program and the address of the shared variables. The level of indirection is an EA variable inside an Effective Address Reference (EAR) structure. All EARs in an SPE program are collected into a Table Of EARs, or TOE. A TOE is declared in a special SPE-ELF `.toe` section.

An EAR is a simple, 16-byte data structure defined as follows:

```
typedef struct elf_toe_entry
{
        ELF64_Addr    ea_value;
        ELF64_Addr    ea_info;     // reserved for future use
} EAR __attribute__ ((aligned (16)));
```

In assembler language, an EAR structure in a `.toe` section can be declared as follows:

```
.section    .toe
.align      7             # align on 128-byte boundary
.global     _EAR_g_mem_obj_1
_EAR_g_mem_obj_1:
        .quad    0        # ea_value
        .quad    0        # ea_info
```

The EAR structures in the SPE-ELF `.toe` sections (and eventually collected into the loadable toe segment) are empty place-holders. They exist so that the SPE linker can bind references to the symbol labels of the EAR structures to absolute LS addresses and thus build a fully resolved

SPE-ELF executable object. At runtime, after the SPE-ELF executable image has been loaded into LS memory but before the SPE program is running, the SPE loader copies the TOE shadow from the PPE-ELF executable over the toe segment in LS. Thereafter, the EAR structures contain valid EAs that have been bound (by the PPE linker) to shared objects in main storage.

Each 16-byte EAR structure is aligned on a 16-byte (quadword) boundary because the entire structure will be loaded into an SPU register at run time. The `ea_value` member is positioned in the structure so that it will be loaded into the register's preferred scalar slot (the most-significant doubleword). Because the EA is already in the preferred scalar slot, an SPU program can transfer the `ea_value` address to the MFC command buffer with minimum overhead.

Using the C-language `spu_mfcdma64` intrinsic, an SPU programmer can use the following code to build a DMA command that copies the shared data object from EA space into LS space:

```
#include <spu_mfcio.h>

extern EAR _EAR_g_mem_obj_1;       // EAR structure to hold the EA of g_mem_obj_1
int g_mem_obj_1;                   // The SPE local (cached) copy of g_mem_obj_1

spu_mfcdma64((volatile void *)&g_mem_obj,              // destination LS
    (unsigned int)(_EAR_g_mem_obj_1.ea_value>>32),     // EA high 32 bits
    (unsigned int)(_EAR_g_mem_obj_1.ea_value),         // EA low 32 bits
    sizeof(g_mem_obj_1),                               // object size in bytes
    0,                                                 // tag id
    MFC_GET);                                          // DMA command
```

14.5.5.1 *Symbol-Name Signatures*

In order to separate ordinary SPE symbols from those representing the references to the EA memory objects, CESOF defines a special symbol name mangling scheme.

In PPE source code, no special coding guidelines are required for the declaration of a shared global data object. In SPE source code, the symbol representing the reference to an EA memory object must be given a name that contains a special signature. This signature must be used only for this purpose. In this JSRE reference implementation, the signature is `_EAR_`. The following is an example of such a declaration:

```
typedef struct elf_toe_entry
{
        ELF64_Addr    ea_value;
        ELF64_Addr    ea_info;         // reserved for future use
} EAR __attribute__ ((aligned (16)));

// the SPE symbol representing the reference to PPE g_mem_obj_1 in EA
const EAR _EAR_g_mem_obj_1;
```

14.5.5.2 *Building the SPE-ELF .toe Section*

When a programmer uses this protocol, the symbol table in an SPE-ELF relocatable object will contain names that have a unique and easily recognizable signature. Assembly language segments can be handcrafted to create this simple array of EAR entries.

Though not provided in the JSRE reference implementation, a simple tool can be built with standard `binutil` commands that can extract these names from the object, create an assembler-language source file for the `.toe` section needed by the relocatable object, and assemble it into a `.toe` object. The relocatable object and the `.toe` object can then be linked into a single relocatable object.

14.5.5.3 *Building a CESOF Object: Embed the SPE Object and Create the TOE Shadow*

When the SPE program has been compiled and linked, with the associated .toe sections, into an SPE-ELF executable object, it is ready to be embedded into a CESOF object, which can then be linked with other PPE-ELF objects to produce an executable suitable for loading and execution by the OS.

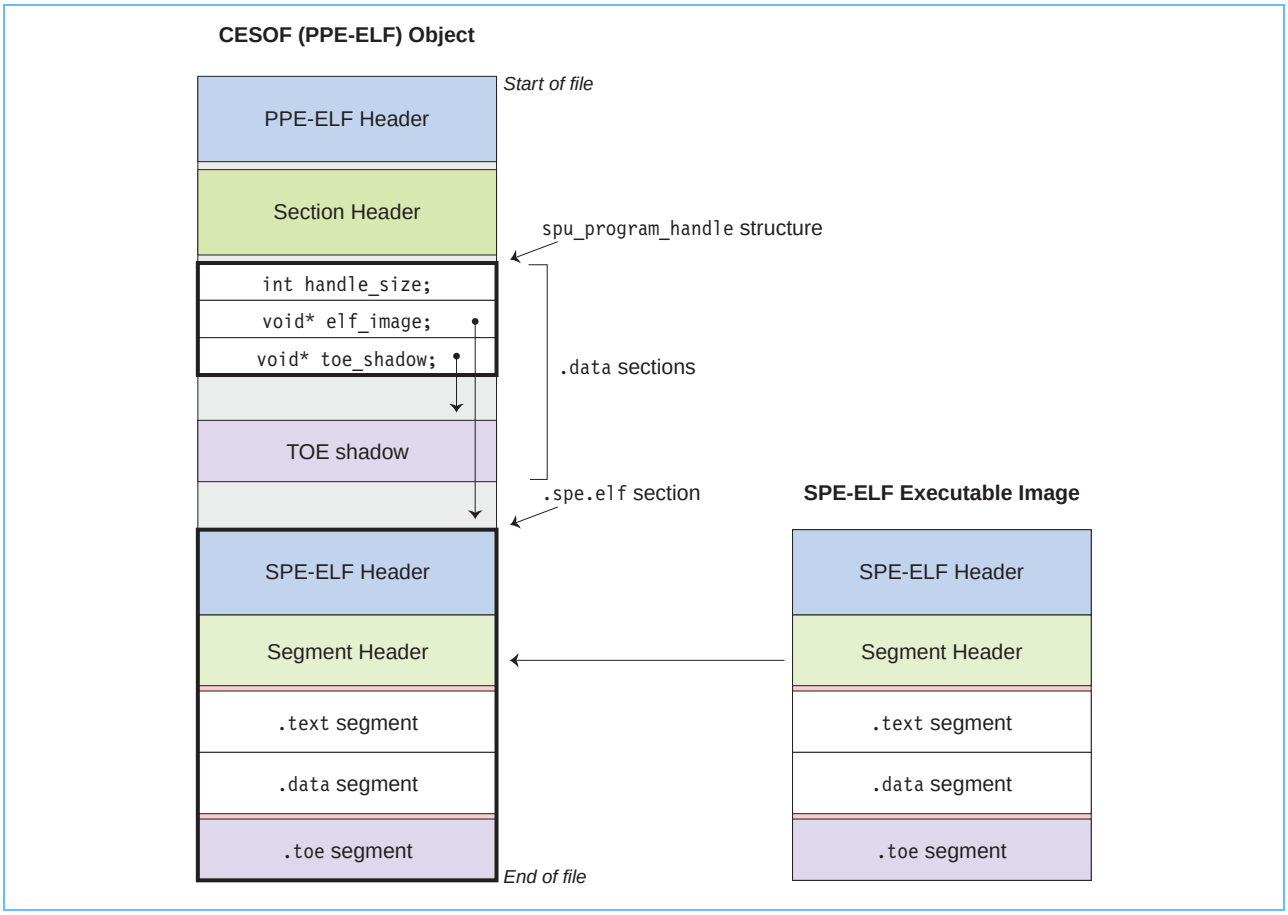
A programmer can build a CESOF relocatable object from the SPE-ELF executable with a tool called “embedspu” in the JSRE reference implementation.

A CESOF object contains three distinct parts:

- The `spe_program_handle` data structure.
- The section containing the embedded SPE-ELF executable object
- The section containing the TOE shadow.

Figure 14-7 shows the layout of a CESOF relocatable object and how an SPE-ELF object is embedded in the `.spe.elf` section.

Figure 14-7. CESOF Object Layout



The following shell-language script illustrates how `embed_spu` can be implemented to construct a CESOF object. This version of the tool builds a CESOF object for a 64-bit PPE OS environment.

```
#!/bin/ksh
#
if [ "x$3" = "x" ]
then
    echo Usage: %0 \<handle_symbol\> \<infile\> \<outfile\>
fi

handle_symbol=$1    # $1 must contain only valid symbol characters

pu-as -o $3 --<<END_xYz
.global ${handle_symbol}
.section .data, "a", @progbits    # definition of the SPE program handle
.align 2
${handle_symbol}:
        .int    20                # 20 bytes for a 64-bit CESOF object
        .quad   _spe_elf_image    # 64-bit pointer
        .quad   _spe_toe_shadow   # 64-bit pointer
```

```
.section .spu.elf, "ax", @progbits
.align 7
_spe_elf_image:
        .incbin "$2"

.section .data, "a", @progbits
.align 7
_spu_toe_shadow:
        `mk_shadow_toe $2`

END_xYz
```

The `embedspu` tool builds assembler source code for an `spe_program_handle` structure, a `.spe.elf` section containing the SPE-ELF executable, and a .data section containing the TOE shadow. The TOE shadow assembler source code is built by the `mk_shadow_toe` tool. The following shows the shell-language source code for a 64-bit version of `mk_shadow_toe`:

```
#!/bin/ksh
# 1. spu-readelf:  dump the symbol table
# 2. egrep:        find the symbols with _EAR_ signature
# 3. sort:         sort the symbols according to their addresses
# 4. awk:          output an EAR entry for each symbol, stripped of signature

spu-readelf -s $1 | egrep '^_EAR_' | sort -k 2 | awk '{ if (index($7, "UND")==0) {\
        print ".quad " substr($8, 6, length($8)); \
        print ".quad  0"; \
}}'
```

After the `mk_shadow_toe` tool creates the TOE shadow code, the `embedspu` tool passes the complete CESOF assembler code to the assembler, which outputs a CESOF relocatable object file. This object can be linked with other PPE-ELF objects to build an executable.

When the PPE-ELF executable is linked together, either statically or dynamically, the TOE shadow will be filled with the bound addresses of shared global data objects, and the SPE loader will write the TOE shadow into LS over the toe segment.

The JSRE reference implementation comes with a more complete implementation of the “embedspu” tool.

14.5.6 Future Software Tool Chain Enhancements for CESOF

The software development process described above requires a programmer to declare shared data objects in SPE source code with a unique signature and use non-standard tools to build the CESOF structures. There are two key deficiencies of this process:

- The SPE programmer must code explicit DMA operations in SPE programs to copy shared global objects into LS and copy them back to main storage.

- The source-language compiler knows nothing of the DMA transfers to and from main storage being performed by the SPE program to cache a shared global object in LS.

One way to overcome these deficiencies is to encode, in an object's declaration, the fact that it is a shared global object. The syntax for such a declaration could be as follows:

```
remote int    g_mem_obj_1 __attribute__((section (".toe")));
remote double g_mem_obj_2 __attribute__((section (".toe")));
```

In these two declarations, the special `section (".toe")` attribute tells the compiler the following:

- Space in LS must be reserved for the objects—an `int` and a `double`—as usual.
- An entry in the special `.toe` section must be built for each EAR that will contain the EA of a shared object in main storage.
- Whenever the program uses an object declared with the `section (".toe")` attribute, the value of the object must be made coherent with the value in main storage. Thus, if the program uses the object as an r-value (a computation source), the compiler must emit an appropriate MFC DMA **get** command that uses the EA from the object's corresponding EAR. If the program uses the object as an l-value (a computation destination), the compiler must emit an appropriate MFC DMA **put** command.

An unsophisticated compiler will emit DMA **get** and/or **put** commands whenever the object is referenced in the program, but a sophisticated compiler could perform software caching of objects by performing live analysis of shared objects and using the synchronization facilities of the MFC, such as lock-line reservation, to reduce the number of MFC DMA commands executed at run time.

14.6 SPE Runtime Loader

Loading cooperative PPE and SPE programs onto a CBE system, from a combined CESOF object or separately, requires loading both PPE and SPE programs into two distinct memory spaces. First, the entire program is loaded into the EA main storage space by the PPE operating system; second, the SPE program is copied, when needed, from EA space into the LS of the SPE that is selected to run the SPE program.

14.6.1 Runtime Loader Overview

A modern operating system, such as Linux, may use a runtime loader to load a binary program into memory for execution. The loader copies or simply maps the program image into the address space of memory that is directly accessible by the processor. In the case of CBE programs, the runtime loader copies or maps code (`.text` in an ELF object), initialized data (`.data` in an ELF object), and uninitialized data (`.bss` in an ELF object) segments into memory and initializes the stack for the process that will contain the running program. There maybe more dynamic linking steps in loading programs linked with dynamic or shared libraries.

After the loader has done its work, it jumps to the entry point of the program in the newly created process.

14.6.2 SPE Runtime Loader Requirements

14.6.2.1 *Static Linking*

An SPE runtime loader can be significantly simpler than a PPE loader because the SPU ABI specifies that programs be statically linked. Static linking results in two major simplifications for an SPE runtime loader: the linker can determine at link time that an SPE program will fit into the SPE's LS, and the SPE loader need not support runtime relocations and dynamic symbol resolution.

14.6.2.2 *Resource Conservation*

An SPE loader that conforms to the framework defined here should use as few resources as possible. The loader should use memory in the LS that can be reclaimed when the loaded SPE application starts; no memory should be permanently allocated to the loader. The loader should ensure that the registers it uses—and that are not otherwise defined as initialized in the ABI—are cleared to '0' before it exits. The loader should take a minimum of execution time.

14.6.2.3 *Specific Loader Actions*

The SPE runtime loader, either executed by PPE or SPE, must accomplish the following actions:

1. Select an available SPE to execute the program, and perform any needed initialization of its memory and access properties.
2. Transfer the SPE program segments, including overlaying the TOE shadow, from EA space to the selected SPE's LS.
3. Initialize the SPU registers according to the requirements of the *SPU Application Binary Interface Specification*.
4. Enable single-step support if needed.
5. Jump to the SPE application program's entry point (typically `crt0` or an equivalent).

Select an Available SPE

This action is handled by the OS running on the PPE or an SPE user-space program. In the latter approach, which has been proven to be a more effective way of loading SPE programs, the PPE first loads a bootstrap loader into the LS, and the bootstrap loader running on the SPE subsequently loads the real SPE program into the LS.

A primary function of the PPE OS is to assign and manage tasks in the system, including tasks for the SPEs. The fundamental responsibilities of the OS for assigning a program to a particular SPE are:

1. Select an SPE to run the program based on the current state of the CBE processor and other application-dependent criteria.
2. Initialize SPE access properties; this includes setting up the SPE's MMU and RMT.
3. Initiate SPE program execution by invoking the SPE bootstrap loader.

After the SPE starts executing, the PPE OS is responsible for managing and controlling the progress of the SPE program.

Transfer Program to LS

The SPE loader runs either on the PPE or SPE using parameters set up by the PPE loader. The SPE loader uses these parameters to locate the SPE program segments (code, data, and possibly a CESOF toe segment from the TOE shadow area) in EA space and copy them to LS space using MFC DMA commands.

Initialize SPE Registers

Certain registers must be initialized before an SPE program is started running. For example, the *Cell Broadband Engine Linux Reference Implementation Application Binary Interface Specification* specifies that R1, the stack pointer (SP), is initialized by the standard `crt0` program entry function to point to the highest quadword address of the SPE's LS. The LS size is 256-KB, so the SP will be initialized to `0x'3FFF0'`. In other environments, the initialization address may be less than `0x'3FFF0'` to accommodate parameters, isolation open area, and so forth.

The *Cell Broadband Engine Linux Reference Implementation Application Binary Interface Specification* also specifies a convention for passing arguments to an SPE program. These arguments include application arguments and application-environment settings. *Section 14-9* on page 400 shows the registers and their initial values.

Enable Single-Step

Single-step execution is required when a debugger is controlling and monitoring an SPU program. Two mechanisms are defined to support SPU single-step operation:

- A privileged-mode register can be set to enable hardware single-step execution. This method is not precise because one or more instructions may be executed before a single-step breakpoint is honored, depending on the fetch and issue rules of the SPU.
- Breakpoints can be set with the SPU **stopd** (stop-and-signal with dependencies) instruction that has the special `0x'3FFF'` signal type. This is a method defined by the CBE ABI and can be as precise as desired by replacing each instruction in turn with the **stopd** instruction. In addition, with this mechanism, breakpoints can be set by either the debugger or the programmer.

Begin Execution

After the SPE program has been copied to LS and the execution environment has been initialized properly, the SPE loader will set the SPE's next program counter (NPC) register to the entry point of the program, typically the entry point to `crt0`, and begin execution. The LS address of the entry point is passed to the SPE loader as a parameter (see *Section 14-11* on page 416).

14.6.3 Example SPE Runtime Loader Framework Definition

This section describes one possible example of an extensible framework for an SPU runtime loader. This framework uses a small bootstrap loader that runs on the SPU and that copies the SPE application program from main storage into LS and initializes the environment according to the ABI. Many other loader strategies are possible.

With this framework, system programmers can build a compatible replacement for the default SPE loader. The default loader may need to be replaced to accommodate extensions to, or re-definitions of, object-file formats, OS environments, and protocols.

The framework described here defines a set of per-binary, per-loader, and per-thread parameters for the SPU loader; each of these parameter sets is described below. This description includes the PPE-initiated and SPE-initiated aspects of the loading process and the interactions between the SPE loader and the application's `crt0` entry function.

14.6.3.1 ***SPE Runtime Loader Parameters***

The SPE loader framework defines data structures for representing and passing parameters to the SPE loader and application program. All of the EAs for the SPE loader are 64-bit pointers. In 32-bit environments, the high-order 32 bits of an EA are cleared to '0'.

The PPE runtime loader is responsible for copying the SPE loader and its parameters into the target SPE's LS. Thus, when the SPE loader begins execution, all three sets of parameters—per-loader, per-binary, and per-thread—are available in LS. Under the definitions discussed below, the three sets of parameters occupy 112 bytes of LS.

Per-Loader Parameters

The per-loader parameters specify the location and size of the SPE bootstrap loader program and the runtime flags the loader should honor. *Table 14-10* explains the per-loader parameters, all of which are 32-bit values.

Table 14-10. SPE Loader Per-Loader Parameters

Parameter	Data Type	Comment
`ldr_ea_hi`	32-bit unsigned int	High-order 32 bits of the EA of the SPE loader in main storage.
`ldr_ea_low`	32-bit unsigned int	Low-order 32 bits of the EA of the SPE loader in main storage.
`ldr_size`	32-bit unsigned int	SPE loader program size in bytes. The maximum value for this parameter is currently set at 384 bytes (24 quadwords).
`ldr_flags`	32-bit unsigned int	Runtime flags for the SPE loader. `SPU_LDR_FLAGS_traced (0x'1')`: if set, the loader enables single-step operation. `SPU_LDR_FLAGS_multi_seg (0x'2')`: if set, the SPE program consists of multiple load segments.

Per-Binary Parameters

The per-binary parameters pass information about the location and size of the SPE program in main storage and in LS. as well as the LS address of the program's entry point. *Table 14-11* explains the per-binary parameters; all the parameters are 32-bit values.

Table 14-11. SPE Loader Per-Binary Parameters

Parameter	Data Type	Comment
`img.size`	32-bit unsigned int	Image size in bytes. If the `SPU_LDR_FLAGS_multi_seg` flag is set in the per-loader `ldr_flags` parameter (see *Table 14-10*), `img.size` indicates the number of segments in the list (see `img.ea_hi` and `img.ea_low`) to be loaded into LS.
`img.ea_hi`	32-bit unsigned int	High-order 32 bits of EA of program image in main storage. If the `SPU_LDR_FLAGS_multi_seg` flag is set in the per-loader `ldr_flags` parameter (see *Table 14-10*), `img.ea_hi` contains the high-order 32-bits of the EA of the list of segments to be loaded into LS.
`img.ea_low`	32-bit unsigned int	Low-order 32 bits of EA of program image in main storage. If the `SPU_LDR_FLAGS_multi_seg` flag is set in the per-loader `ldr_flags` parameter (see *Table 14-10*), `img.ea_low` contains the low-order 32-bits of the EA of the list of segments to be loaded into LS.
`img.dst_ls`	32-bit unsigned int	LS destination address for the SPE program image segment.
`entry`	32-bit unsigned int	LS address of the SPE program entry point (typically the start of `crt0`).
`lslr`	32-bit unsigned int	LS limit setting; this value is copied from the `spu_env` structure in the SPU Environment Note in the SPE-ELF Program Header; see *Section 14.2.2.4*.
`elf_ea_hi`	32-bit unsigned int	High-order 32 bits of the EA of the SPE-ELF executable in main storage.
`elf_ea_low`	32-bit unsigned int	Low-order 32 bits of the EA of the SPE-ELF executable in main storage.

In the simplest case, the SPE program executable contains a single, combined, loadable segment containing the `.text`, `.data`, and `.bss` segments. In this case, the `img.ea_hi`, `img.ea_low`, and `img.size` parameters specify the region of main storage to be copied into LS.

In other cases, particularly for an executable using the capabilities of CESOF, the SPE-ELF executable will have more than one load segment. For multiple load segments, the `img.ea_hi` and `img.ea_low` parameters specify the EA of a list in main storage. The `img.size` parameter specifies the number of elements in the list. The SPE loader uses the list to copy each of the loadable segments from main storage to LS memory.

In the case of a CESOF executable, the list will contain at least two elements: an element for the combined segment that contains the `.text`, `.data`, and `.bss` segments of the SPE program and an element for the toe segment. When the PPE loader builds the list elements, it will replace the main-storage address of the toe segment with the address of the TOE shadow section. Consequently, the SPE loader will overwrite the toe segment in LS with the TOE shadow. This fulfills the CESOF runtime requirement of replacing the `.toe` with the TOE shadow.

Since the framework must support even more sophisticated SPE loaders, the EA in main storage of the complete SPE-ELF executable image is available in the `elf_ea_hi` and `elf_ea_low` parameters.

Per-Thread Parameters

The per-thread parameters are arguments passed to the SPE program because the running instance of the program is an OS thread. These parameters are passed to satisfy the ABI register-initialization requirements discussed in *Section 14.6.2.3* on page 413, and are explained in *Table 14-9* on page 400.

The `spu_lscsa` in R6 may be used by the PPE OS when it needs to preempt the SPE thread, or the thread itself may use the area to perform a light-weight SPE context save. An SPE thread may perform a context save in response to a privileged-attention event raised by the PPE OS (see *Section 18* on page 463), or the thread may save context as a means to communicate system-call parameters from LS or SPU registers to the PPE OS.

14.6.3.2 *PPE-Initiated Actions for SPE Loading*

The first steps in loading an SPE program into an SPE LS for execution include loading the SPE-ELF executable image from the CESOF object and loading the SPE loader, called `spu_ld.so` in this discussion, into main storage. These steps are shown graphically in *Figure 14-8* on page 417.

Figure 14-8. PPE OS Loads CESOF Object and spu_ld.so into EA Space

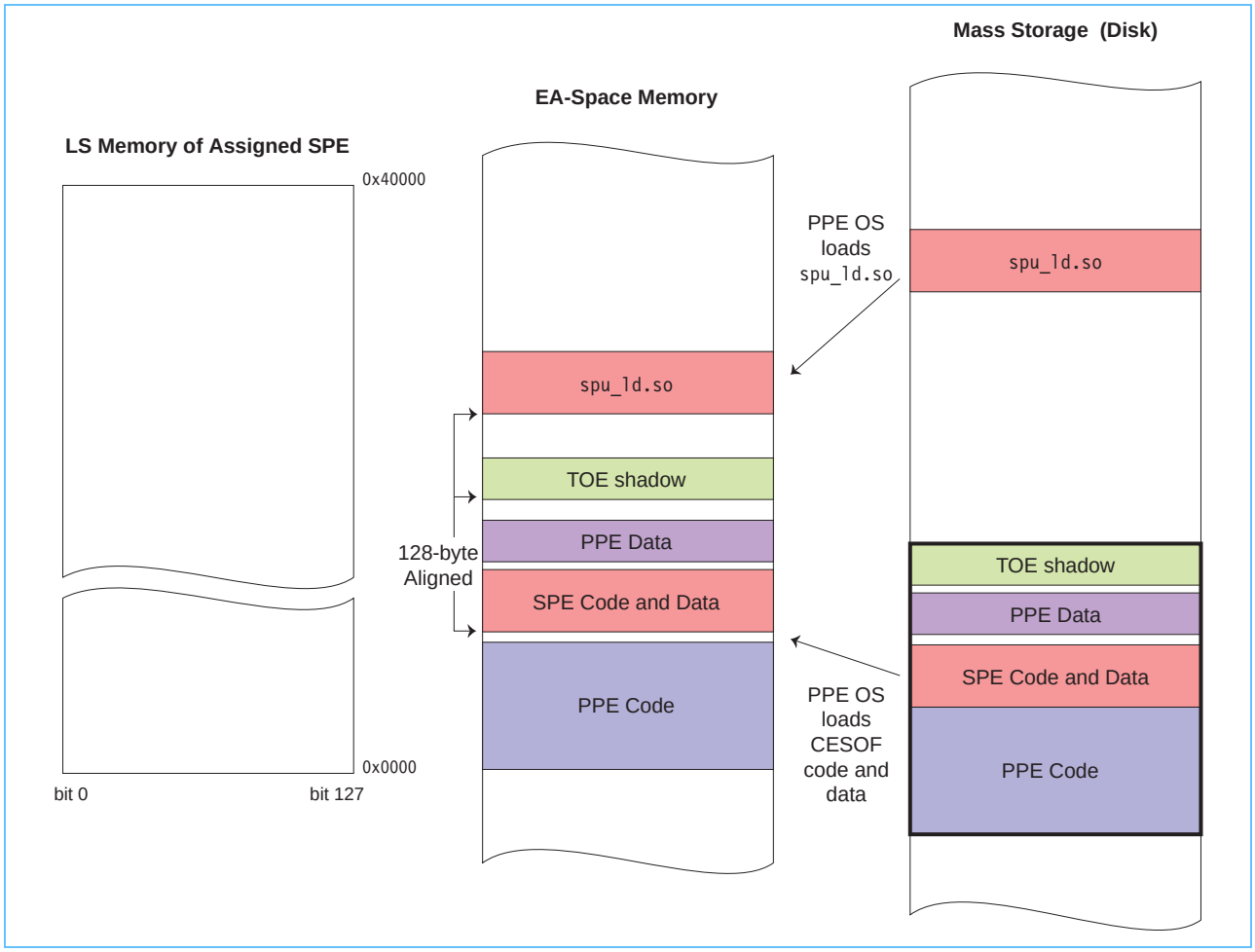
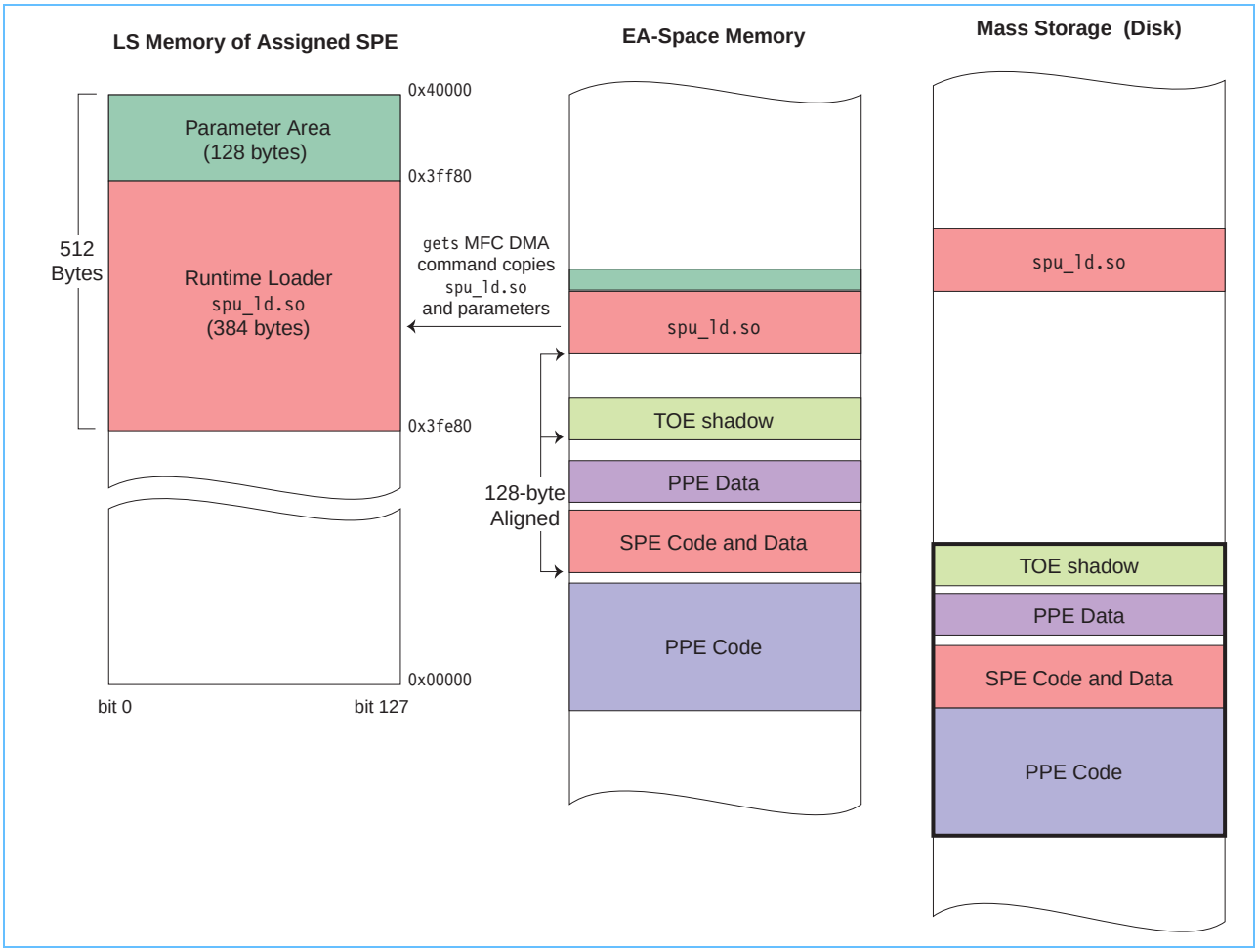


Figure 14-9 on page 418 shows the location and sizes of the loader and its parameters after they have been copied to LS.

Figure 14-9. PPE OS Loads `spu_ld.so` and Parameters into SPE LS



Before the PPE loader can copy the SPE loader and set it running, the PPE OS must assign the SPE program to an available SPE. The assigned SPE's state, including its LS and registers, must be cleaned (for example, initialized to zero), and the SPE's access and memory properties (such as the contents of the MMU and RMU) must be properly initialized.

If the PPE loader detects that it is loading an SPE program that has a toe segment, it must create a list in main storage of loadable segments that will cause the SPE loader to copy the TOE shadow from the PPE-ELF wrapper to LS instead of the toe segment from the SPE-ELF executable. Creating the segment list and copying the TOE shadow over the toe segment are the major aspects of the runtime support required by the CESOF specification.

Next, the PPE loader sets the SPE's NPC register to the loader entry point, which, as shown in *Table 14-12* on page 419, is `SPU_LDR_PROG_start` with the value (`LS_SIZE` – `512`). Since `LS_SIZE` is 256 KB, the NPC will be set to `0x'3FE00'`.

The next step is invoking the SPE loader with the proper parameters. The PPE loader copies the SPE boot-strap loader and its parameters to LS according to the values shown in *Table 14-12*.

Table 14-12. LS Locations and Sizes of SPE Loader and Parameters

Attribute	Value	Comment
SPU_LDR_PROG_start	LS_SIZE – 512	The SPE loader is copied by the PPE loader into LS at the address of the start of the last 512-byte (32-quadword) block of implemented LS memory. For the CBE processor, for which LS_SIZE is 0x'40000', this address is 0x'3FE00'.
SPU_LDR_PROG_size	384	The current framework allows the SPE loader to occupy up to 384 bytes (24 quadwords) of LS starting at SPU_LDR_PROG_start.
SPU_LDR_PARAMS_start	LS_SIZE – 128	The SPE loader parameters are copied by the PPE loader into LS at the address of the start of the last 128-byte (8-quadword) block of implemented LS memory. For the CBE processor, for which LS_SIZE is 0x'40000', this address is 0x'3FF80'.
SPU_LDR_PARAMS_size	128	The current framework allows the SPE loader parameters to occupy up to 128 bytes (8 quadwords) of LS starting at SPU_LDR_PARAMS_start.

The PPE loader copies the SPE loader parameters to LS by initiating a DMA **get** command in the SPE's associated MFC (see *Section 19* on page 505 for details on DMA commands). Next, the PPE loader copies the SPE loader code to LS by initiating a DMA **gets** command. The **gets** command copies the code and then, after the DMA transfer has completed, starts the SPE's SPU running at the address in the NPC register, which is set to 0x'3FE00'.

At this point, the state of LS is shown in *Figure 14-9* on page 418, and the work of the PPE loader is finished; the SPE bootstrap loader completes the remaining steps in the loading process. A principle advantage of dividing the work of SPE program loading between the PPE loader and the SPE loader is the fact that after the PPE loader enqueues the **get** and **gets** commands, it need not wait for them to complete; the PPE loader can continue with other aspects of the PPE loading process or terminate and free the PPE for other tasks.

14.6.3.3 ***SPE-Initiated Actions for SPE Loading***

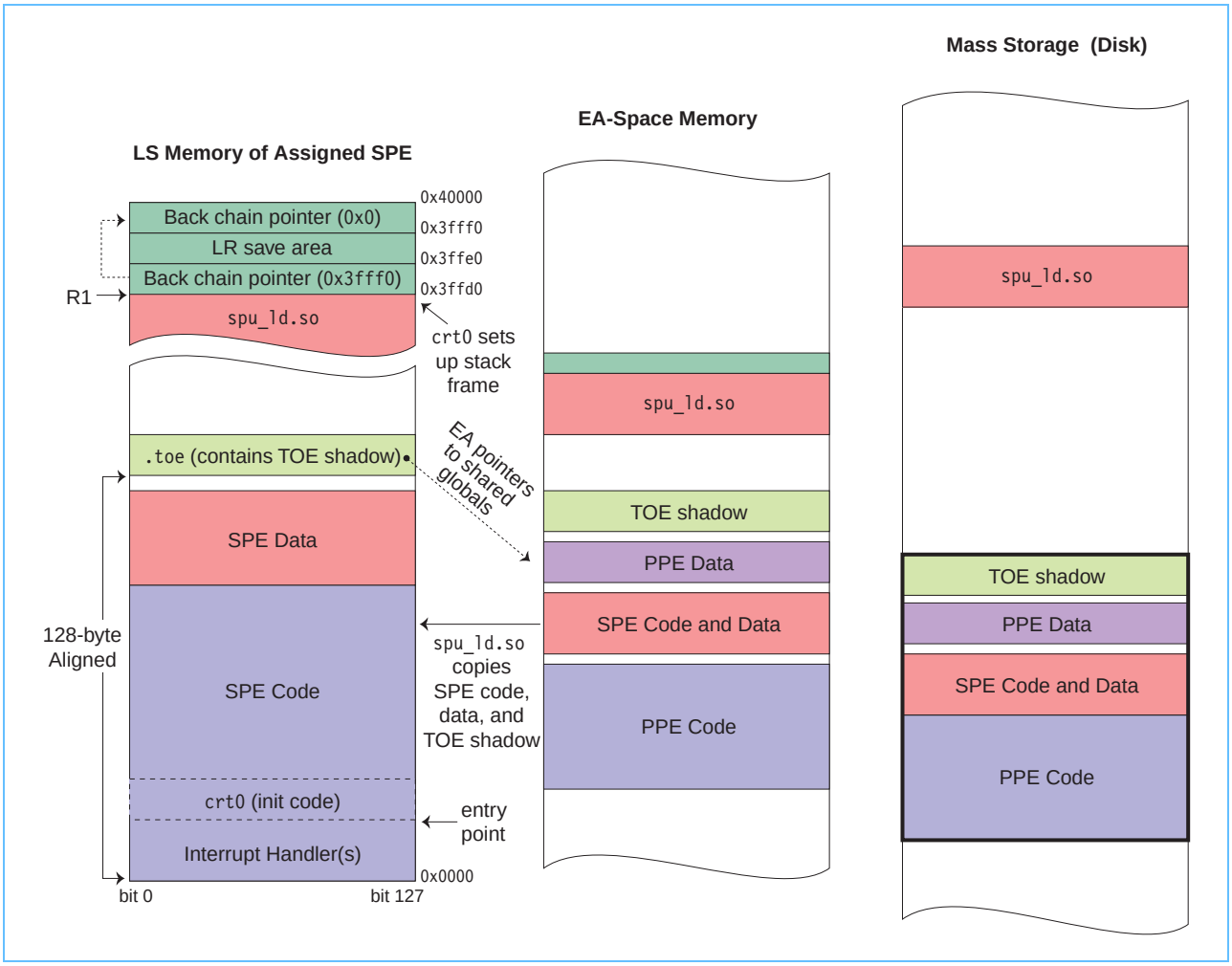
The SPE bootstrap loader must copy the SPE program segments to LS, initialize registers, possibly enable single-step mode (see *Section 14.6.2.3* on page 413), and finally start the SPE executing the loaded program.

Copy SPE Program Segments to LS

The sizes and locations of the program segments to be copied are contained in the per-binary section of the parameter area. The parameters can specify either a single loadable segment or a list of segments. If a list of segments is needed to properly load the SPE program, the PPE loader is responsible for setting up the list in main storage. See *Section 14.6.3.1* on page 415 for additional information on per-binary parameters. As summarized in *Section* on page 412, copying the TOE shadow segment is a key part of CESOF runtime support.

In the case of loading a CESOF executable, the final state of LS—with the SPE application loaded and the TOE shadow copied in place of the toe segment—is shown in *Figure 14-10* on page 420.

Figure 14-10. `spu_ld.so` Loads SPE Code and TOE Shadow



Initialize Registers

See *Section 14-9* on page 400 for details on how R1 (stack pointer) and R2 (stack size) are set. The SPE loader initializes registers R3 through R5 according to the per-thread parameters. See *Section 14.6.2.3* on page 413 for more information.

Enable Single Step

This SPE loader framework implements single-stepping and debugger breakpoints with the **stop** instruction. The return code of the **stop** instruction for a breakpoint is `0x'3FFF'`.

If the `SPU_LDR_FLAGS_traced` bit is set to '1' in the per-loader `ldr_flags` parameter, the SPE program will be executed in single-step mode. To support single-step mode, the SPE loader must execute the **stop** instruction just before the loader branches to the SPE program's entry point. When controlling software, such as an SPE debugger, takes control after this breakpoint is raised, this software can set additional breakpoints or otherwise modify the SPE program state.

Begin SPE Program Execution

The last step is to transfer control to the SPE program's entry point. The address of the entry point is found in the `entry` parameter of the per-binary parameter area. The `entry` parameter typically contains the starting address of the program's `crt0` function. If single-step mode is enabled, the loader executes a breakpoint **stop** instruction before it branches to `crt0`; otherwise it simply branches to `crt0`.

14.6.3.4 *SPE Loader Interactions with `crt0`*

The PPE loader clears to zero the assigned SPE's registers and LS before the SPE loader and its parameters are copied to LS. When the SPE loader has completed its work, the SPE application program is in LS and registers are initialized according to the ABI requirements.

All other LS regions and SPU registers contain zero, except for the last 512 bytes (32 quadwords) of LS, which is where the SPE loader and parameters are located. Although `crt0` need not clear regions of LS that contain uninitialized data segments (the PPE loader already cleared all of LS), it may be possible that some systems will be designed to have `crt0` clear the 512-byte area used by the SPE loader. Explicit clearing of the SPE loader area is probably not necessary, however, because `crt0` is responsible for initializing the SPE stack pointer to the highest LS memory address and setting up the initial stack frame. Consequently, function calls in the SPE application will overwrite the SPE loader and parameters. In any case, it is unlikely that `spu_ld.so` and its parameters will contain any sensitive information.

14.7 SPE Execution Environment

This section specifies low-level system information and conventions that are used by the operating system. Some of these conventions are defined by the ABI and are standard for all systems. Others are specific to the Linux operating system environment and are indicated as such. For specific additional details, consult the operating-environments documentation for your specific operating system.

14.7.1 Signal Types for the SPE Stop-and-Signal Instruction

The **stop** instruction contains a 14-bit signal-type field. When **stop** is executed, the signal type is written to bits 0 through 13 of the SPE status register. This value is used to indicate why a program stopped. The signal-type values have been partitioned by Linux in the following way:

- Signal-type values with the most-significant bit cleared to '0' are reserved for application use.
- Signal-type values with the most-significant bit set to '1' are reserved for runtime or privileged services.

Table 14-13 describes the reserved signal type values.

Table 14-13. SPE-ELF Reserved Signal Type Values

Signal-Type Value	Description
`0x'0000'`	Data executed as an instruction.

`0x'2000'..0x'20FF'`	Linux return from `main()` or `exit()`. Return or `exit()` status is encoded in the least-significant byte of the stop-and-signal value:
	`exit (EXIT_SUCCESS) == 0x'2000'`
	`exit (EXIT_FAILURE) == 0x'2001'`

`0x'2100'..0x'21FF'`	Linux PPE assisted library calls. See *Cell Broadband Engine Linux Reference Implementation Application Binary Interface Specification* for additional details.
`0x'3FFE'`	Stack overflow detected.
`0x'3FFF'`	Debugger breakpoint.

15. Power and Thermal Management

The CBE power-management mechanism provides a means for limiting the amount of power dissipated by a component by limiting its degree of activity. There are five states for power management. Typically, the more aggressive the power-management state, the more time is required to enter and exit the state. These states apply to each of the various processing elements and other logic blocks of the CBE processor. Each component can be individually set to a power-management state. The power-management facilities provide privileged software with the ability to manage power while meeting the real-time demands of a system. The CBE processor implements a subset of the power-management states defined by the *Cell Broadband Engine Architecture*.

The CBE digital thermal-management mechanism provides a means for monitoring and controlling the temperature within the CBE processor. During the execution of an application, the temperature of areas within the CBE processor can rise. Left unchecked, the temperature could rise above the maximum specified junction temperature leading to improper operation or physical damage.

Both the power-management and thermal-management facilities are part of the CBE processor's Pervasive Logic, which provides control, monitoring, and debugging functions.

15.1 Power Management

Table 15-1 summarizes the CBE processor's power-management states. Details on each state are given below.

Table 15-1. Power-Management States

Power Management State	Description
CBE Slow State	In CBE Slow state, clock frequency is lowered. A lower clock frequency results in greater potential for performance degradation, and lower power consumption. For more information, see *Section 15.1.1 CBE Slow State*.
PPE Pause (0) State	PPE Pause (0) state is a clock-gating that occurs when both PPE threads are shut down. PPE code execution is suspended during this state. For more information, see *Section 15.1.2 PPE Pause (0) State*.
SPE State Retained and Isolated (SRI) State	In SPE SRI state, all access to the component is inhibited. The state remaining on the component is retained. The component must be prepared by privileged software or hardware to maintain system integrity. The component does not make forward progress. For more information, see the security documentation, available under NDA.
SPU Pause State	In SPU Pause state, SPU code execution can be suspended to reduce power and stop forward progress until resumed. MMIO registers and the LS can still be accessed. For more information, see *Section 15.1.3 SPU Pause State*.
MFC Pause State	In MFC Pause state, MFC command queue operations can be suspended to reduce power and to stop forward progress until resumed. For more information, see *Section 15.1.4 MFC Pause State*.
Active State	In Active state, no power-management functions are enabled. This is the state of maximum performance and maximum power-use.

15.1.1 CBE Slow State

The main clock frequency of the CBE processor (NClk) can be changed dynamically in the CBE Slow state. The functions of the CBE processor are not affected in this state; only the performance is affected. Certain time-base settings are guaranteed to be invariant over all frequency changes. See *Section 13 Time Base and Decrementers* on page 375 for details. CBE Slow state can be run in conjunction with any other power-management state.

15.1.1.1 *Configuration-Ring Settings*

CBE Slow has one related Configuration-Ring setting, `slow_mode_delay_setting`. The recommended setting is x'1F'. See the *Cell Broadband Engine Hardware Initialization Guide* for a description of the Configuration Ring.

15.1.1.2 *PLL Start*

By default, the CBE processor starts at the maximum frequency. However, if the phase-locked loop (PLL) is configured to not start at the maximum frequency, an extra step is required before scaling the NClk frequency. This required step is to write `PMCR[Override]` = '1' and `PMCR[Curr_Slow_n]` = PLL frequency division. This can be done with one MMIO write. Software must not initiate a frequency scale until a read of `PMSR[BE_Slow_n]` reflects the PLL frequency division.

15.1.1.3 *Scaling the CBE Core Clock (NClk) Frequency*

CBE Slow changes the CBE Core Clock (`NClk`) frequency (see *Section 13.2.3 Time-Base Frequency* on page 377) to a target frequency rate (n). Changing NClk requires a compensating factor (command-pacing rate) in the MIC unit to prevent buffer overruns. This is required since the MIC clock (MiClk) runs separately and does not change with this procedure. An initial setting of the `MIC_Slow_Fast_Timer_n` register sets the command-pacing rate during the frequency change. This value is constant for all frequencies and needs to be set up only once. The `MIC_Slow_Next_Timer_n` register sets the command pacing rate after the frequency change; this value is different for each target frequency (n). After all pacing rates are set, a write to the `PMCR[Next_Slow_n]` register field initiates a frequency change. No writes to `MIC_Slow_Fast_Timer_n`, `MIC_Slow_Next_Timer_n`, or `PMCR[Next_Slow_n]` occur until the status of `PMSR[BE_Slow_n]` reflects the target frequency.

To dynamically change to a supported frequency, perform the following steps:

> ***Note:*** *If the PLL is not configured to start at the maximum frequency, perform the step described in Section 15.1.1.2 before performing these steps.*

1. See *Table 15-2* on page 425 for the desired target frequency and corresponding recommended register values.
2. Software sets `MIC_Slow_Fast_Timer_0[8:19]` and `MIC_Slow_Fast_Timer_1[8:19]` to x'7FC'. (Software only has to write these registers once.)
3. Software sets the new `MIC_Slow_Next_Timer_0[16:27]`, and `MIC_Slow_Next_Timer_1[16:27]` to the value indicated in *Table 15-2* for the desired target frequency.
4. Software sets the new frequency `PMCR[Next_Slow_n]` to the value indicated in *Table 15-2* for the desired target frequency.

5. Software waits for a read of `PMSR[BE_Slow_n]` to reflect the updated frequency change. Status must reflect the updated frequency before attempting another frequency change.

Table 15-2 shows the register updates corresponding to desired target frequencies for CBE Slow state. Some CBE processors may not be certified for execution in all of the slow-state frequencies shown in *Table 15-2*. Consult the processor's data sheet for supported frequencies.

Table 15-2. Register Updates for CBE Slow State

`PMCR[Next_Slow(n)]` [n = 0:7]	`MIC_Slow_Fast_Timer_n`[8:19] [n = 0,1]	`MIC_Slow_Next_Timer_n`[16:27] [n = 0,1]
n = 0 Frequency = Max	x‘7FC’	x‘240’
n = 1 Frequency = Max/2	x‘7FC’	x‘268’
n = 2 Frequency = Max/3	x‘7FC’	x‘29C’
n = 3 Frequency = Max/4	x‘7FC’	x‘2D0’
n = 4 Frequency = Max/5	x‘7FC’	x‘300’
n = 5 Frequency = Max/6	x‘7FC’	x‘334’
n = 6 Frequency = Max/8	x‘7FC’	x‘39C’
n = 7 Frequency = Max/10	x‘7FC’	x‘3FC’

Note: `MIC_Slow_Next_Timer_n` table data assumes a NCLK/MiClk ratio of 2:1. For an equation with different ratios see in the *Cell Broadband Engine Registers* document.

For details about the registers referenced in this chapter, see the *Cell Broadband Engine Registers* document.

15.1.2 PPE Pause (0) State

PPE Pause (0) state requires both PPE threads to be shut down and pause state-control is enabled. Cache coherency is maintained during this state. PPE Pause (0) state is exited with interrupts, some of which can be masked individually. PPE Pause (0) state can be run in conjunction with any other power-management state.

15.1.2.1 *Enabling PPE Pause (0) State*

Control of PPE Pause (0) is disabled by default. A one-time write to `PMCR[PPE_pause]` = ‘1’ is required to enable the state.

15.1.2.2 *Entering PPE Pause (0) State*

Each PPE thread must be shut down for the PPE Pause (0) state to be enabled. After both threads are shut down, the PPE Pause (0) state is entered as long as no interrupt is pending. See *Section 10 PPE Multithreading* on page 295 for details about resuming and suspending a thread.

15.1.2.3 *Exiting Pause State*

PPE Pause (0) state exits by one of the following types of interrupts: External, Decrementer, Thermal Management, or System Error. Pause control masking is possible for External and Decrementer interrupts. See *Section 15.1.2.4 Pause Interrupt Resume Masks* for information on masking.

It is possible to exit the PPE Pause (0) state and not wake a thread, because there are separate PPE thread controls. See *Section 10 PPE Multithreading* on page 295 for details about resuming and suspending a thread.

Note: *A pending Hypervisor Decrementer (HDEC) interrupt prevents entering the PPE Pause (0) state. However, once the state is activated, an HDEC interrupt does not exit the PPE Pause (0) state.*

15.1.2.4 *Pause Interrupt Resume Masks*

The External and Decrementer interrupts (per thread) can be selectively masked to determine whether they cause an exit from PPE Pause (0) state. The mask must be set before both threads are shut down by setting a combination of any mask bits located at `TSCR[wdec0]`, `TSCR[wdec1]`, or `TSCR[wext]`. This mask is only for interrupts to exit PPE Pause (0) state; it is possible to exit PPE Pause (0) state and not wake a thread because these masks are different. See *Section 10 PPE Multithreading* on page 295 for details.

15.1.3 SPU Pause State

SPU Pause state can be run in conjunction with the CBE Slow, PPE Pause (0), and MFC Pause states. The MMIO registers and the LS can still be accessed in this state. Cache coherency is maintained in this state.

Software pauses (suspends) an SPU by writing `MFC_SR1[S]` = ‘0’. Software resumes an SPU by writing `MFC_SR1[S]` = ‘1’.

15.1.4 MFC Pause State

MFC Pause state can be run in conjunction with the CBE Slow, PPE Pause (0), and SPU Pause states. The MMIO registers and the LS can still be accessed in this state.

Software pauses (suspends) an MFC’s command queues by writing `MFC_CNTL[Sc]` = ‘1’. Cache coherency is maintained in this state. Software resumes an MFC’s command queues by writing `MFC_CNTL[Sc]` = ‘0’.

15.2 Thermal Management

The CBE processor’s digital thermal-management unit consists of a Thermal-Management Control Unit (TMCU) and ten distributed Digital Thermal Sensors (DTSs). One sensor is located in each of the eight SPEs, one is located in the PPE, and one is adjacent to the linear thermal diode. The linear thermal diode is an on-chip diode that calculates temperature. These sensors are adjacent to areas within the associated unit that typically experience the greatest rise in temperature during the execution of most applications.

The TMCU monitors feedback from each of the sensors. If the temperature of a sensor rises above a programmable point, the TMCU can be configured to cause an interrupt to the PPE and dynamically throttle the execution of the associated PPE or SPE. The throttling is accomplished by stopping and running the PPE or SPE for a programmable number of cycles. The interrupt allows privileged software to take corrective action while dynamic throttling (a hardware device) attempts to keep the temperature below a programmable level without software intervention. Privileged software must set the throttling level equal to or below the settings recommended in the CBE processor's datasheet.

If dynamic throttling or privileged software does not effectively manage the temperature and the temperature continues to rise, the CBE processor's clocks are stopped when the temperature reaches a thermal overload temperature defined by the Configuration Ring data. The thermal overload feature protects the CBE processor from physical damage. Recovery from this condition requires a hard reset.

Note: *The temperature of the region monitored by the digital thermal sensors is not necessarily the hottest point within the associated PPE or SPE.*

15.2.1 Thermal-Management Operation

In the TMCU, each DTS provides a current temperature-detection signal. This signal indicates that the temperature is equal to or below the current temperature-detection range set by the TMCU. The TMCU uses the state of these signals to continually track the temperature of the PPE's or each SPE's digital thermal sensor. As the temperature is tracked, the TMCU provides the current temperature as a numeric value that represents the temperature within the PPE or SPE. The mapping between this numeric value and the temperature is provided in *Table 15-3*. Internal calibration storage is set in manufacturing to calibrate the individual sensors.

Table 15-3. Digital Temperature-Sensor Encoding (Page 1 of 2)

Thermal-Sensor Temperature Encoding			
6-bit Encode	Temperature Range	6-bit Encode	Temperature Range
0	$temp < 65.0°C$	17	$97.0°C \le temp < 99.0°C$
1	$65.0°C \le temp < 67.0°C$	18	$99.0°C \le temp < 101.0°C$
2	$67.0°C \le temp < 69.0°C$	19	$101.0°C \le temp < 103.0°C$
3	$69.0°C \le temp < 71.0°C$	20	$103.0°C \le temp < 105.0°C$
4	$71.0°C \le temp < 73.0°C$	21	$105.0°C \le temp < 107.0°C$
5	$73.0°C \le temp < 75.0°C$	22	$107.0°C \le temp < 109.0°C$
6	$75.0°C \le temp < 77.0°C$	23	$109.0°C \le temp < 111.0°C$
7	$77.0°C \le temp < 79.0°C$	24	$111.0°C \le temp < 113.0°C$
8	$79.0°C \le temp < 81.0°C$	25	$113.0°C \le temp < 115.0°C$
9	$81.0°C \le temp < 83.0°C$	26	$115.0°C \le temp < 117.0°C$
10	$83.0°C \le temp < 85.0°C$	27	$117.0°C \le temp < 119.0°C$
11	$85.0°C \le temp < 87.0°C$	28	$119.0°C \le temp < 121.0°C$
12	$87.0°C \le temp < 89.0°C$	29	$121.0°C \le temp < 123.0°C$
13	$89.0°C \le temp < 91.0°C$	30	$123.0°C \le temp < 125.0°C$

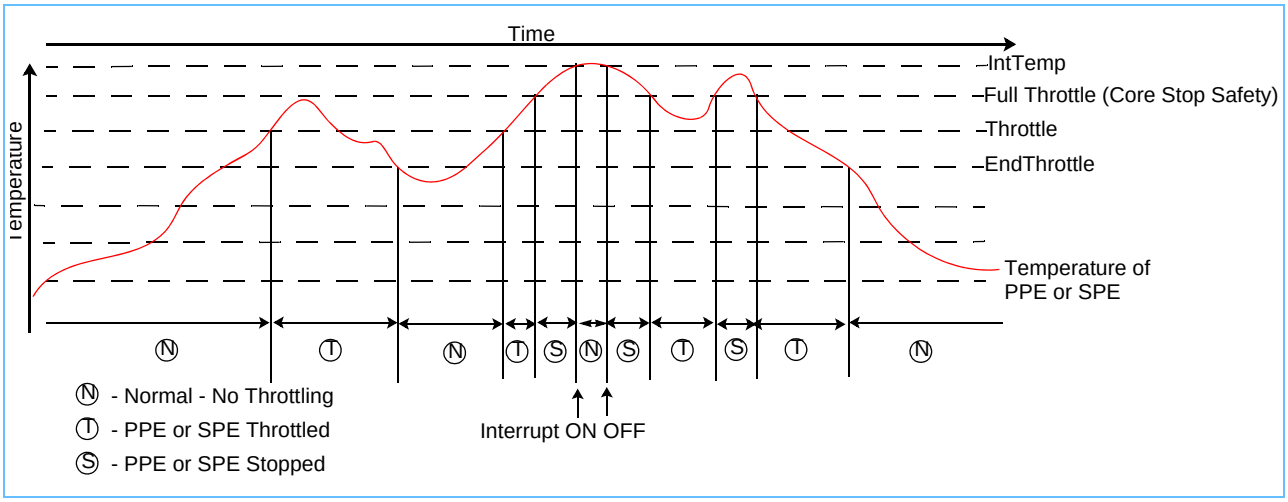
Table 15-3. Digital Temperature-Sensor Encoding (Page 2 of 2)

Thermal-Sensor Temperature Encoding			
6-bit Encode	Temperature Range	6-bit Encode	Temperature Range
14	$91.0°C \le temp < 93.0°C$	31-63	$125.0°C \le temp$
15	$93.0°C \le temp < 95.0°C$		
16	$95.0°C \le temp < 97.0°C$		

The TMCU can be configured to cause an interrupt to the PPE and dynamically throttle the execution of the PPE or an SPE. The TMCU compares the numeric value representing the temperature to a programmable interrupt temperature and a programmable throttle point. If the temperature is within the programmed interrupt temperature range, an external interrupt is generated to the PPE, if enabled. In addition, a second programmable interrupt temperature can cause the assertion of an Attention signal to a system controller. See *Section 15.2.5 Thermal Sensor Interrupt Registers* on page 429 for more information on the interrupt capability of the TMCU. If the temperature is equal to or above the throttling point, the TMCU throttles the execution of the PPE or an SPE by starting and stopping the PPE or SPE dynamically. Software can control the ratio and frequency of the throttling using the Dynamic Thermal-Management registers. See *Section 15.2.6 Dynamic Thermal-Management Registers* on page 432 for more information.

Figure 15-1 on page 428 illustrates the temperature and the various points at which interrupts and dynamic throttling occur. In this figure, the PPE or SPE is running normally; there is no throttling in the regions marked with an “N”. When the temperature reaches the throttle point, the TMCU starts throttling the execution of the associated PPE or SPE. The regions in which the throttling occurs are marked with a “T”. When the temperature of the PPE or SPE drops below the end throttle point, the execution returns to normal operation. If, for any reason, the temperature continues to rise and reaches a temperature at or above the full throttle point, the PPE or SPE is stopped until the temperature drops below the full throttle point. Regions where the PPE or SPE is stopped are marked with an “S”. Stopping the PPE or SPEs when the temperature is at or above the full throttle point is referred to as the Core Stop Safety. In this illustration, the interrupt temperature is set above the throttle point; therefore, software is notified if the PPE or SPE is ever stopped for this condition, provided that the Thermal Interrupt Mask register (`TM_ISR`) is set to active, allowing the PPE or SPE to resume during a pending interrupt.

Figure 15-1. Digital Thermal-Sensor Throttling



Note: *If dynamic throttling is disabled, privileged software must manage the thermal condition. Not managing the thermal condition can result in improper operation of the associated PPE or SPE or a thermal shutdown by the thermal-overload function.*

15.2.2 Configuration-Ring Settings

Thermal management uses the following Configuration-Ring settings:

- `MinStopPPE` specifies the minimum stop time for the PPE. Suggested value: x'8'.
- `MinStopSPE` specifies the minimum stop time for the SPEs. Suggested value: x'8'.
- `CFG_TO` specifies the temperature which will cause the Thermal Overload signal to be asserted and the clocks to be stopped. Suggested value: x'1F'.
- `SenSamp`Time specifies how often a sensor will be sampled. Suggested value: x'01'.
- `DigFiltDly` specifies the delay time for the sensor-output's digital filter. Suggested value: x'0'.

For a description of the Configuration Ring, see the *Cell Broadband Engine Hardware Initialization Guide*.

15.2.3 Thermal Registers

See the *Pervasive MMIO Registers* section of the *Cell Broadband Engine Registers* document for bit definitions of the thermal-management MMIO registers mentioned in the following sections. See the *Cell Broadband Engine Hardware Initialization Guide* for a description of the Configuration Ring.

15.2.4 Thermal Sensor Status Registers

The Thermal Sensor Status registers consist of the Thermal Sensor Current Temperature Status registers (`TS_CTSR1` and `TS_CTSR2`) and the Thermal Sensor Maximum Temperature Status registers (`TS_MTSR1` and `TS_MTSR2`). These registers allow software to read the current temperature of each DTS, determine the highest temperature reached during a period of time, and cause an interrupt when the temperature reaches a programmable temperature. System software should mark the real address pages containing the Thermal Sensor registers as hypervisor privileged.

`TS_CTSR1` and `TS_CTSR2` contain the encoding for the current temperature of each DTS. `TS_MTSR1` and `TS_MTSR2` contain the encoding for the maximum temperature reached for each sensor from the time of the last read of these registers. See the *Cell Broadband Engine Registers* specification for a definition of the encoding. The length of a sample period is controlled by the `SenSamp-Time` configuration field, as described in the *Cell Broadband Engine Hardware Initialization Guide*.

15.2.5 Thermal Sensor Interrupt Registers

The Thermal Sensor Interrupt registers control the generation of a thermal-management interrupt to the PPE. This set of registers consists of the Thermal Sensor Interrupt Temperature registers (`TS_ITR1` and `TS_ITR2`), the Thermal Sensor Interrupt Status register (`TS_ISR`), the Thermal Sensor Interrupt Mask register (`TS_IMR`), and the Thermal Sensor Global Interrupt Temperature register (`TS_GITR`).

TS_ITR1, TS_ITR2, and TS_GITR contain the encoding for the temperature that causes a thermal-management interrupt to the PPE. See the *Cell Broadband Engine Registers* specification for a definition of the encoding.

When the temperature encoding in TS_CTSR1 and TS_CTSR2 for a sensor is greater than or equal to the corresponding sensor's interrupt temperature encoding in TS_ITR1 and TS_ITR2, the corresponding status bit (TS_ISR[Sx]) is set. When the temperature encoding in TS_CTSR1 and TS_CTSR2 for any sensor is greater than or equal to the global interrupt temperature encoding in TS_GITR, the corresponding status bits (TS_ISR[Gx]) are set.

If any TS_ISR[Sx] bit is set and the corresponding mask bit (TS_IMR[Mx]) is also set, a thermal-management interrupt signal is asserted to the PPE. If any TS_ISR[Gx] bit is set and the corresponding mask bit (TS_IMR[Cx]) is also set, a thermal-management interrupt signal is asserted to the PPE. To clear the interrupt condition, privileged software should clear any corresponding mask bits in TS_IMR to '0'.

To enable a thermal-management interrupt, privileged software must ensure that the temperature is below the interrupt temperature for the corresponding sensors and then perform the following sequence; enabling an interrupt when the temperature is not below the interrupt temperature can result in an immediate thermal-management interrupt's being generated:

1. Write a '1' to the corresponding status bit in TS_ISR.
2. Write a '1' to the corresponding mask bit in TS_IMR.

15.2.5.1 *Thermal Sensor Interrupt Temperature Registers*

The Thermal Sensor Interrupt Temperature registers (TS_ITR1 and TS_ITR2) contain the interrupt temperature level for the sensors located in the SPEs, PPE, and adjacent to the linear thermal diode. The encoded interrupt temperature levels in this register are compared to the corresponding interrupt temperature encoding in TS_CTSR1 and TS_CTSR2. The results of these comparisons are used to generate a thermal-management interrupt. Each sensor's interrupt temperature level is independent.

15.2.5.2 *Thermal Sensor Global Interrupt Temperature Register*

In addition to the independent interrupt temperature levels set in TS_ITR1 and TS_ITR2, the Thermal Sensor Global Interrupt Temperature register (TS_GITR) contains a second interrupt temperature level. This level applies to all sensors in the CBE processor. The encoded global interrupt temperature level in this register is compared to the current temperature encoding for each sensor. The results of these comparisons are used to generate a thermal-management interrupt.

The intent of the global interrupt temperature is to provide an early indication to a temperature rise in the CBE processor. Privileged software and the system controller can use this information to start actions to control the temperature, for example, increasing the fan speed, rebalancing the application software across units, and so on.

15.2.5.3 *Thermal Sensor Interrupt Status Register*

The Thermal Sensor Interrupt Status register (TS_ISR) identifies which sensors meet the interrupt conditions. An interrupt condition refers to a particular condition that each TS_ISR bit has that, when met, makes it possible for an interrupt to occur. An actual interrupt is only presented to the PPE if the corresponding mask bit is set in the Thermal Sensor Interrupt Mask register (TS_IMR).

TS_ISR contains three sets of status bits—the Digital Sensor Global Threshold Interrupt status bit (TS_ISR[Gx]), the Digital Sensor Threshold Interrupt status bit (TS_ISR[Sx]), and the Digital Sensor Global Below Threshold Interrupt status bit (TS_ISR[Gb]).

Hardware sets TS_ISR[Sx] when the temperature encoding for a sensor in TS_CTSR1 and TS_CTSR2 is greater than or equal to the corresponding sensor's interrupt temperature encoding in TS_ITR1 and TS_ITR2 and the corresponding direction bit TM_IMR[Bx] = '0'. Also, hardware sets TS_ISR[Sx] when the temperature encoding for a sensor in TS_CTSR1 and TS_CTSR2 is below the corresponding sensor's interrupt temperature encoding in TS_ITR1 and TS_ITR2 and the corresponding direction bit TM_IMR[Bx] = '1'.

Hardware sets TS_ISR[Gx] when any participating sensor's current temperature is greater than or equal to that of TS_GITR and TS_IMR[BG] is cleared to '0'. The individual TS_ISR[Gx] bits indicate which individual sensors meet these conditions.

Hardware sets TS_ISR[Gb] when all of the participating sensors in TS_IMR[Cx] have a current temperature below that of TS_GITR and the TS_IMR[BG] is set to '1'. Since all participating sensors must have a current temperature below that of TS_GITR, there is only one status bit (TS_ISR[Gb]) for a global below-threshold interrupt condition.

Once a status bit (TS_ISR[Sx], [Gx], or [Gb]) is set to '1', this state is maintained until reset to '0' by privileged software. Privileged software resets a status bit to '0' by writing a '1' to the corresponding bit in TS_ISR.

15.2.5.4 *Thermal Sensor Interrupt Mask Register*

The Thermal Sensor Interrupt Mask register (TS_IMR) contains two fields for individual sensors and multiple fields for global interrupt conditions. An interrupt condition refers to a particular condition that each TS_IMR bit has that, when met, makes it possible for an interrupt to occur. An actual interrupt is only presented to the PPE if the corresponding mask bit is set.

The two TS_IMR Digital Thermal Threshold Interrupt fields for individual sensors are TS_IMR[Mx] and TS_IMR[Bx]. The TS_IMR[Mx] mask bits prevent an interrupt status bit from generating a thermal-management interrupt to the PPE. The TS_IMR[Bx] directional bits set the temperature direction for the interrupt condition above or below the corresponding temperature in TS_ITR1 and TS_ITR2. Setting TS_IMR[Bx] to '1' sets the temperature for the interrupt condition to be below the corresponding temperature in TS_ITR1 and TS_ITR2. Clearing TS_IMR[Bx] to '0' sets the temperature for the interrupt condition to be equal to or above the corresponding temperature in TS_ITR1 and TS_ITR2.

The TS_IMR fields for the global interrupt conditions are TS_IMR[Cx], TS_IMR[BG], TS_IMR[Cgb], and TS_IMR[A]. The TS_IMR[Cx] mask bits prevent global threshold interrupts and select which sensors participate in the global below threshold interrupt condition. The TS_IMR[BG] directional

bit selects the temperature direction for the global interrupt condition. The TS_IMR[Cgb] mask bit prevents global below threshold interrupts. TS_IMR[A] asserts an attention to the system controller.

Setting TS_IMR[BG] to '1' sets a temperature range for the global interrupt condition to occur when the temperatures of all the participating sensors set in TS_IMR[Cx] are below the global interrupt temperature level. Clearing TS_IMR[BG] to '0' sets a temperature range for the global interrupt condition to occur when the temperature of any of the participating sensors is greater than or equal to the corresponding temperature in TS_GITR. If TS_IMR[A] is set to '1', an attention is asserted when any TS_IMR[Cx] bit and its corresponding status bit (TS_ISR[Gx]) are both set to '1'. Also, an attention is asserted when TS_IMR[Cgb] and TS_ISR[Gb] are both set to '1'.

A thermal-management interrupt is presented to the PPE when any TS_IMR[Mx] bit and its corresponding status bit (TS_ISR[Sx]) are both set to '1'. A thermal-management interrupt is also generated when any TS_IMR[Cx] bit and its corresponding status bit (TS_ISR[Gx]) are both set to '1'. Also, a thermal-management interrupt is presented to the PPE when TS_IMR[Cgb] and TS_ISR[Gb] are both set to '1'.

15.2.6 Dynamic Thermal-Management Registers

The Dynamic Thermal-Management registers contain parameters for controlling the execution throttling of the PPE or an SPE. This set of registers contains the Thermal-Management Control registers (TM_CR1 and TM_CR2), the Thermal-Management Throttle Point register (TM_TPR), the Thermal-Management Stop Time registers (TM_STR1 and TM_STR2), the Thermal-Management Throttle Scale register (TM_TSR), and the Thermal-Management System Interrupt Mask register (TM_SIMR).

TM_TPR sets the throttle point for the sensors. Two independent throttle points can be set in TM_TPR[ThrottlePPE/ThrottleSPE]—one for the PPE and one for the SPEs. Also contained in this register are temperature points for exiting throttling and stopping the PPE or SPEs. Execution throttling of the PPE or an SPE starts when the temperature is equal to or above the throttle point. Throttling ceases when the temperature drops below the temperature to exit throttling (TM_TPR[EndThrottlePPE/EndThrottleSPE]). If the Temperature reaches the full throttle or stop temperature (TM_TPR[FullThrottlePPE/FullThrottleSPE]), the execution of the PPE or SPE is stopped. TM_CR1 and TM_CR2 are used to control the throttling behavior.

TM_STR1, TM_STR2, and TM_TSR are used to control the frequency and amount of throttling. When the temperature reaches the throttle point, the corresponding PPE or SPE is stopped for the number of clocks specified by the corresponding scale value in the TM_TSR. The PPE or SPE is then allowed to run for the number of clocks specified by the run value in the TM_STR1 and TM_STR2 registers times the corresponding scale value. This sequence, shown below, continues until the temperature falls below the exit throttling (TM_TPR[EndThrottlePPE/EndThrottleSPE]).

TM_SIMR is used to select which interrupts exit throttling of the PPE while the interrupt is pending.

A simplified throttling process for the SPEs is illustrated below. The value TM_Config[MinStopSPE] comes from the Configuration Ring. All other values below are from Thermal-Management MMIO registers.

```
if  (TS_CTSR1[Cur(x)] >= TM_TPR[Throttle]) {
       /* where x is the PPE or SPE and 0 ≤ x ≤ 7) */
```

```
    While (TS_CTSR1[Cur(x)] >= TM_TPR[EndThrottleSPE]) {
        if (TS_CTSR1[Cur(x)] >= TM_TPR[FullThrottleSPE]){
            stop SPE(x)
        } else {
            spcstopunit = (TM_Config[MinStopSPE] * TM_TSR[ScaleSPE])
            stop SPE(x) for (spcstopunit * TM_STR1[StopCore(x)]) cycles
            run SPE(x) for (spcstopunit * (32 - TM_STR1[StopCore(x)])) cycles
        }
    }
}
run SPE(x)
```

The process for throttling the PPE is similar to that for the SPE. The following is the routine for PPE throttling:

```
if  (TS_CTSR2[Cur(8)] >= TM_TPR[Throttle]) & (No_Interrupts_Pending) {
        /* No_Interrupts_Pending depends on the setting of TM_SIMR */
    While (TS_CTSR2[Cur(8)] >= TM_TPR[EndThrottlePPE]) {
        if (TS_CTSR2[Cur(8)] >= TM_TPR[FullThrottlePPE]){
            /* TM_TPR[FullThrottlePPE] should be set at a high enough temperature to
             * avoid stopping PPE
             */
            stop PPE
        } else {
            PPEstopunit = (TM_Config[MinStopPPE] * TM_TSR[ScalePPE])
            stop PPE for (PPEstopunit * TM_STR2[StopCore(8)]) cycles
            run PPE for (PPEstopunit * (32 - TM_STR2[StopCore(8)])) cycles
        }
    }
}
run PPE
```

The preceding code sample shows a simplified throttling process. The actual hardware throttling implementation must also account for updates of the Thermal-Management and Thermal-Sensor registers, interrupts, and interface protocols.

Privileged software should set the full throttling temperature `TM_TPR[FullThrottlePPE]`) high enough to avoid the condition where hardware stops the PPE. Stopping the PPE prevents an interrupt from being processed.

15.2.6.1 *Thermal-Management Control Registers*

The Thermal-Management Control registers (`TM_CR1` and `TM_CR2`) set the throttling mode for the PPE or each SPE independently. The control bits are split between two registers. There are five different modes that can be set for the PPE or each SPE independently:

- Dynamic throttling disabled (including the Core Stop Safety).
- Normal operation (dynamic throttling and the Core Stop Safety are enabled).
- PPE or SPE is always throttled (Core Stop Safety is enabled).

- Core Stop Safety disabled (dynamic throttling enabled and the Core Stop Safety are disabled).
- PPE or SPE is always throttled and Core Stop Safety disabled.

Privileged software should set control bits to normal operation for PPE or SPEs that are running applications or operating systems. If the PPE or an SPE is not running application code, privileged software should set the control bits to disabled. The “PPE or SPE is always throttled” modes are intended for application development. These modes are useful to determine if the application can operate under an extreme throttling condition. Allowing the PPE or an SPE to execute with either the dynamic throttling or Core Stop Safety disabled should only be permitted when privileged software actively manages the thermal events.

15.2.6.2 *Thermal-Management System Interrupt Mask Register*

The Thermal-Management System Interrupt Mask register (`TM_SIMR`) controls which PPE interrupts cause the thermal-management logic to temporarily stop throttling the PPE. Throttling is temporarily suspended for both threads while the interrupt is pending, regardless of the thread targeted by the interrupt. When the interrupt is no longer pending, throttling can resume as long as throttle conditions still exist. Throttling of the SPEs is never exited based on a system interrupt condition. The PPE interrupt conditions that can override a throttling condition are:

- External.
- Decrementer.
- Hypervisor Decrementer.
- System Error.
- Thermal Management.

15.2.6.3 *Thermal-Management Throttle Point Register*

The Thermal-Management Throttle Point register (`TM_TPR`) contains the encoded temperature points at which execution throttling of the PPE or an SPE begins and ends. This register also contains encoded temperature points at which the PPE or SPE execution is fully throttled.

The values in this register are used to set three temperature points for changing between the three thermal-management states: Normal Run (N), PPE or SPE Throttled (T), and PPE or SPE Stopped (S). Independent temperature points are supported for the PPE and the SPEs.

When the encoded current temperature of a sensor in `TS_CTSR` is equal to or greater than the throttle temperature (`ThrottlePPE/ThrottleSPE`), execution throttling of the corresponding PPE or SPE begins, if enabled. Execution throttling continues until the encoded current temperature of the corresponding sensor is less than the encoded temperature to end throttling (`EndThrottlePPE/EndThrottleSPE`). As a safety measure, if the encoded current temperature is equal to or greater than the full throttle point (FullThrottlePPE/FullThrottleSPE), the corresponding PPE or SPE is stopped.

15.2.6.4 *Thermal-Management Stop Time Registers*

The Thermal-Management Stop Time registers (`TM_STR1` and `TM_STR2`) control the amount of throttling applied to a specific PPE or SPE in the thermal-management throttled state. The values in this register are expressed as a percentage of time that the PPE or an SPE is stopped versus the time that it is run (`CoreStop(x)`/32). The actual number of clocks (NClks) that the PPE or an SPE stops and runs is controlled by the Thermal-Management Scale register (`TM_TSR`).

15.2.6.5 *Thermal-Management Throttle Scale Register*

The Thermal-Management Throttle Scale register (`TM_TSR`) controls the actual number of cycles that the PPE or an SPE stops and runs during the thermal-management throttle state. The values in this register are multiples of a Configuration-Ring setting, `TM_Config[MinStopSPE]`. The actual number of stop and run cycles is calculated by the following equation:

```
SPE Run and Stop Time:
----------------------

SPE_StopTime = (TM_STR1[StopCore(x)] * TM_Config[MinStopSPE]) * TM_TSR[ScaleSPE]
SPE_RunTime = (32 - TM_STR1[StopCore(x)]) * TM_Config[MinStopSPE]) * TM_TSR[ScaleSPE]

PPE Run and Stop Time:
----------------------

PPE_StopTime = (TM_STR2[StopCore(8)] * TM_Config[MinStopPPE]) * TM_TSR[ScalePPE]
PPE_RunTime = (32 - TM_STR2[StopCore(8)]) * TM_Config[MinStopPPE]) * TM_TSR[ScalePPE]
```

Note: *The run and stop times can be altered by interrupts and privileged software writing various Thermal-Management registers.*

16. Performance Monitoring

The CBE processor provides extensive performance-monitoring facilities that assist performance analysis, as well as provide application- and system-optimization features that include:

- Debugging, analyzing, and optimizing processor-architecture features.
- Profiling the behavior of the memory hierarchy and the interaction of multiple address spaces, as well as tuning system and application algorithms to optimize scheduling, partitioning, and structuring for tasks and data.
- Real-time application-tuning by monitoring bandwidth use and other resource-management behavior.

The performance-monitoring facilities can count and log data on over 400 types of internal events. Up to eight types of events can be monitored concurrently. The facilities support the monitoring of classes of instructions selected by an instruction-matching mechanism, the random selection of instructions for detailed monitoring, and count start/stop event pairs that exceed a selected time-out threshold.

The facilities are designed for tuning a broad range of software, including:

- Numerically intensive floating-point applications.
- Fixed-point applications.
- System software.

The facilities give clear visibility to the details of instruction execution, loads and stores, the behavior of caches throughout the CBE processor, the entire virtual-memory architecture—including effective-to-real address translation (ERAT), translation-lookaside buffers (TLBs), data-prefetch streaming, and the effects of large-page support—and the activity of the EIB, MIC, IOC, and several other CBE components that can have large effects on application performance.

The performance monitor facility provides eight 16-bit counters. Counters can be paired (combined) to function as four 32-bit counters for count values to be collected per measurement interval. The interval is timed by a 32-bit programmable interval timer. Performance information can be obtained on signals from either a single island or multiple islands. The performance monitor has the following notable features:

- Counts the number of cycles active or inactive over an interval, or the number of positive or negative transitions over an interval.
- The interval is programmable through an interval timer.
- Performance-monitor signals from the PPE can be masked according to the state of the PPE (for example, hypervisor mode).
- Performance-monitor conditions, such as an interval timeout, can be programmed to cause an interrupt.

The performance monitor counters can be frozen until a selected start count qualifier occurs. The counters can continue to count up until the occurrence of a stop count qualifier. The counters can also be frozen when a counter overflows. An initial count other than ‘0’ can also be applied to the counters to give them a programmable threshold. When the performance monitor counters are frozen, the interval timer is also frozen. See the *Cell Broadband Engine Registers* document for additional details.

16.1 How It Works

Access to the registers used in performance monitoring requires hypervisor privilege. The performance-counting facilities share most of the resources used for hardware debug of the CBE processor. Because performance monitoring tries to maximize the reuse of the debug resources, some debug facilities are reinterpreted for performance-monitoring use. For example, the “trigger bus” and the “event bus” have specific meanings for debug, but for performance monitoring the two buses are regarded as extra buses that are used to route performance-monitoring events.

The basic flow of performance monitoring goes as follows: Each unit in CBE processor generates various signals (per-event, per-cycle, or both). These signals are selected by writing certain MMIO registers. Each signal is connected to one or more bus lines. The bus lines are routed to individual counters. The counters are then connected to a trace array which records the counts periodically.

The debug bus is the transport mechanism for delivering signals from the various CBE units to the performance monitor. This bus is shared by the performance monitor and the trace-logic analyzer. The performance monitor is disabled when the trace-logic analyzer is enabled. Signals to be counted are identified with tags, which indicate whether the signal is useful for counting cycles or edges.

Performance signals are sent to the performance counters by means of the 128-bit debug bus (+ 11 special bits). The 128-bit bus is divided into four words, and most events can be sent to one of two different words of the bus. For example, the signals for a PPU thread’s instruction-unit events can go on either word 2 or word 3, the signals for an MFC’s L2-cache events can go on word 0 or word 1, and the signals for an IOC event can go on word 0 or word 2. Debug-specific signals are reinterpreted for performance monitoring and can use the extra 11 bits normally used for debug.

16.2 Events (Signals)

The performance-monitoring facilities can monitor many types of events from all major units on the CBE processor. See *Appendix C Performance Monitor Signals* on page 781 for the full list of verified events (also called signals).

16.3 Performance Counters

There are eight 16-bit counters (which can be combined to make four 32-bit counters) and a trace-array collection and storage mechanism. The trace-array mechanism uses both internal and external storage resources. The counters can be set up to count events from either PowerPC Processor Unit (PPU) thread, several of the PowerPC Processor Storage Subsystem (PPSS) internal logic blocks, any of the eight SPUs, any of the eight MFCs and their internal logic blocks, the Element Interconnect Bus (EIB), the Memory Interface Controller (MIC), the Bus Interface Controller (BIC), and the I/O Interface Controller (IOC).

Each counter can monitor one event type at a time, and the events are of three kinds:

- *Cycles*—The cycles in which an event are active or inactive.
- *Event Edges*—The edge transitions between two event states. The counters can be frozen until a specified trigger occurs before they begin counting the desired event.

- *Event Cycles*—The cycles until a specified trigger-event occurs, at which times the counters are frozen.

One set of counters can be configured as 16-bit *count qualifiers*, such that the start and stop count of the counting can be specified. This makes it possible to control when the counting of an event starts and/or stops.

After the counters are set up and begin recording events, the counters can be read interactively by a user program. For example, a user program running in a logical partition can make a system call to its supervisor software, which will then make a hypervisor call that reads the appropriate registers and returns the collected data to the user program.

For details, see the documentation for your tools that are layered on the CBE processor's performance monitoring features.

16.4 Trace Array

The trace array contains 1,024 128-bit entries. It is used to store the count values at the end of each interval. When the trace array is filled, a PPE interrupt can be set to occur. The data can be transferred from the trace array to main storage either by simple PPE MMIO reads (128-bits at a time) or by DMA transfers. Count data can also be transferred to external memory to capture thousands of intervals worth of performance data.

Counted events are automatically saved into the trace array according to a specified interval. All performance counters are reset to '0' after each trace-array logging interval. A mechanism is provided for monitoring the operation of the interval timer.

17. SPE Channel and Related MMIO Interface

17.1 Introduction

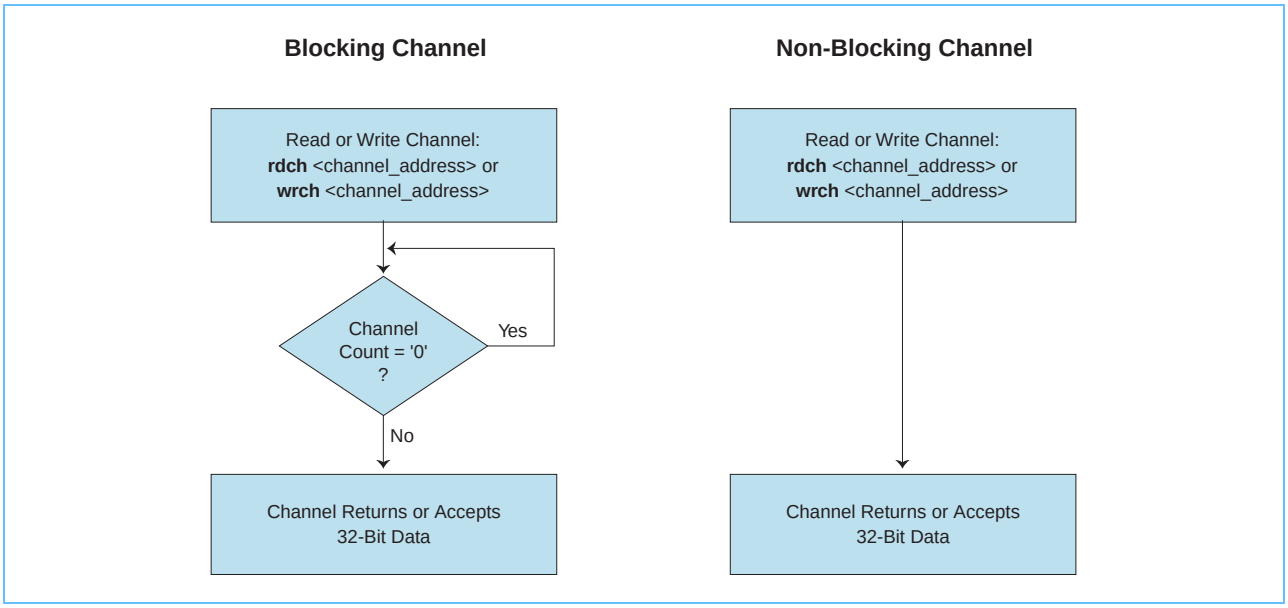
17.1.1 An SPE's Use of its Own Channels

An SPE communicates with the PPE and other SPEs and devices through its channels. Channels are unidirectional interfaces for sending and receiving variable-size (up to 32-bit) messages with the PPE and other SPEs, and for sending commands (such as DMA transfer commands) to the SPE's associated MFC. Each SPE has its own set of channels.

SPE software accesses channels with three instructions: read channel (**rdch**), write channel (**wrch**), and read channel count (**rchcnt**). These instructions enqueue MFC commands into the SPE's MFC SPU Command Queue for the purpose of initiating DMA transfers, querying DMA and SPU status, sending mailbox and signal-notification messages for synchronization of tasks, and accessing auxiliary resources such as the SPE's decrementer.

The channels are architecturally defined as blocking or non-blocking. When SPE software reads or writes a non-blocking channel, the operation executes without delay. However, when SPE software reads or writes a blocking channel, the SPE may stall for an arbitrary length if the associated channel count (which is its remaining capacity) is '0'. In this case, the SPE will remain stalled until the channel count becomes '1' or more, as shown in *Figure 17-1* on page 441, or the SPE is interrupted. The stalling mechanism allows an SPE to minimize the power consumed by message-based synchronization of tasks, because fewer logic gates are switching during the stall.

Figure 17-1. SPE Access to Its Own Channels



17.1.2 Access to Channel Functions by the PPE and other SPEs

Software on the PPE and other devices (including other SPEs) can gain access to some of the facilities an SPU can access through its channel interface—including the passing of mailbox and signal-notification messages—by accessing associated MMIO registers in the main-storage effective address (EA) space. Unlike an SPE's accesses to it channels, MMIO accesses are never blocked.

MMIO accesses enqueue MFC commands into the SPE's MFC Proxy Command Queue—a different queue than the MFC SPU Command Queue used by the SPE, and independent thereof. Although the PPE can use this means of message-passing for synchronization of tasks, such as storage, the PPE more commonly uses memory-barrier instructions (*Section 20* on page 551) available in the *PowerPC Architecture* instruction set.

17.1.3 Channel Characteristics

Each SPE channel is either read-only or write-only and is either blocking or non-blocking. Each channel has a corresponding capacity (maximum message entries), and count (remaining capacity). This channel count decrements whenever a channel instruction (**`rdch`** or **`wrch`**) is issued to the corresponding channel, and the count increments whenever an operation associated with that channel completes. Thus, a channel count of '0' means "full" for write-only channels and "empty" for read-only channels. A blocking channel will cause the SPE to stall when the SPE reads or writes the channel and the channel count is '0'.

Table 17-1 summarizes the characteristics of the four channel types.

Table 17-1. SPE Channel Types and Characteristics

Channel Type	Valid Instructions	Blocking	Description
Read	**rdch** **rchcnt**	Non-Blocking	**rdch** returns without delay; **rdch** will never stall. **rchcnt** always returns '1'.
		Blocking	**rdch** returns only if the channel count is not '0'; **rdch** stalls if queue empty. **rchcnt** returns number of items waiting in queue.
Write	**wrch** **rchcnt**	Non-Blocking	**wrch** always puts data without delay; **wrch** will never stall. **rchcnt** always returns '1'.
		Blocking	**wrch** puts data only if the channel count is not '0'; **wrch** stalls if queue is full. **rchcnt** returns number of queue entries available to accept data.

The *Synergistic Processor Unit Instruction Set Architecture* specifies that a **rdch** or **wrch** instruction to an invalid channel causes the SPE to stop on or after the instruction. However, the CBE-processor implementation differs from the SPU ISA in that **rdch** instructions to reserved channels or valid write channels return '0's and **wrch** instructions to reserved channels or valid read channels have no effect.

Reading or writing an unimplemented (reserved) channel does not cause an illegal channel instruction interrupt. The **`rchcnt`** instruction responds differently depending on the type of channel it is directed at:

- **`rchcnt`** returns '0' when directed to an unimplemented (reserved) channel.

- `rchcnt` returns ‘1’ when directed to an implemented, non-blocking channel.
- `rchcnt` returns the capacity (write-only channel) or occupancy (read-only channel) for an implemented, blocking channel.

17.1.4 Channel Summary

The key features of the SPE channel operations include:

- All operations on a given channel are unidirectional (they can be only read or write operations for a given channel, not bidirectional).
- Accesses to channel-interface resources through MMIO addresses do not stall.
- Channel operations are done in program order.
- Channel read operations to reserved channels return ‘0’s.
- Channel write operations to reserved channels have no effect.
- Reading of channel counts on reserved channels returns ‘0’.
- Channel instructions use the 32-bit preferred slot in a 128-bit transfer.

Table 17-2 on page 443 lists the SPE channels, and their corresponding MMIO registers, that each SPE supports. The EA column under the MMIO Register heading gives the offset in hexadecimal from the base address for the SPE. For further details, see the *Cell Broadband Engine Architecture*.

The function of channels whose channel mnemonic in *Table 17-2* begins with the prefix `SPU_` is confined to the SPE; channels whose channel mnemonic begins with the prefix `MFC_` access the SPE’s MFC for operations outside the SPE, such as DMA transfers, synchronization, mailbox messages, and signal notification.

Most channels are 32 bits wide; only two are 16 bits wide. The *SPU C/C++ Language Extensions* define intrinsics for reading and writing channels using 32-bit (scalar `unsigned int`) or 128-bit (vector) operands. When setting a 16-bit channel, only the least-significant 16 bits of a 32-bit scalar operand is used.

Table 17-2. SPE Channels and Associated MMIO Registers (Page 1 of 3)

SPE Channel #	Name	Channel Interface[1] (MFC SPU Command Queue)					MMIO Register Interface[1] (MFC Proxy Command Queue)				
		Mnemonic	Max. Entries	Blocking	R/W	Width (bits)	Offset From Base	Mnemonic	Max. Entries	R/W	Width (bits)
SPU Events											
0	SPU Read Event Status	SPU_RdEventStat	1	yes	R	32	—	—	—	—	—
1	SPU Write Event Mask	SPU_WrEventMask	1	no	W	32	—	—	—	—	—
2	SPU Write Event Acknowledgment	SPU_WrEventAck	1	no	W	32	—	—	—	—	—
SPU Signal Notification											
3	SPU Signal Notification 1	SPU_RdSigNotify1	1	yes	R	32	x‘1400C’	SPU_Sig_Notify_1	1	R/W	32
4	SPU Signal Notification 2	SPU_RdSigNotify2	1	yes	R	32	x’1C00C’	SPU_Sig_Notify_2	1	R/W	32

Table 17-2. SPE Channels and Associated MMIO Registers (Page 2 of 3)

SPE Channel #	Name	Channel Interface[1] (MFC SPU Command Queue)					MMIO Register Interface[1] (MFC Proxy Command Queue)				
		Mnemonic	Max. Entries	Blocking	R/W	Width (bits)	Offset From Base	Mnemonic	Max. Entries	R/W	Width (bits)
—	SPU Configuration[2,9]	—	—	—	—	—	x'04078'	SPU_Cfg	1	R/W	64
5	Reserved	—	—	—	—	—	—	—	—	—	—
6	Reserved	—	—	—	—	—	—	—	—	—	—
SPU Decrementer											
7	SPU Write Decrementer	SPU_WrDec	1	no	W	32	—	—	—	—	—
8	SPU Read Decrementer	SPU_RdDec	1	no	R	32	—	—	—	—	—
MFC Multisource Synchronization											
9	MFC Write Multisource Synchronization Request	MFC_WrMSSyncReq	1	yes	W	32	x'00000'	MFC_MSSync	1	R/W	64
10	Reserved	—	—	—	—	—	—	—	—	—	—
SPU and MFC Read Mask											
11	SPU Read Event Mask	SPU_RdEventMask	1	no	R	32	—	—	—	—	—
12	MFC Read Tag-Group Query Mask	MFC_RdTagMask	1	no	R	32	—	—	—	—	—
SPU State Management											
13	SPU Read Machine Status	SPU_RdMachStat	1	no	R	32	—	—	—	—	—
14	SPU Write State Save-and-Restore	SPU_WrSRR0	1	no	W	32	—	—	—	—	—
15	SPU Read State Save-and-Restore	SPU_RdSRR0	1	no	R	32	—	—	—	—	—
MFC Command Parameters											
16	MFC Local-Storage Address	MFC_LSA	1	no	W	32	x'03004'	MFC_LSA	1	W	32
17	MFC Effective Address High	MFC_EAH	1	no	W	32	x'03008'	MFC_EAH	1	W	32
18	MFC Effective Address Low or List Address	MFC_EAL	1	no	W	32	x'0300C'	MFC_EAL	1	W	32
19	MFC Transfer Size or List Size	MFC_Size	1	no	W	16[3]	x'03010'	MFC_Size	1	W	16 H
20	MFC Command Tag Identification	MFC_TagID	1	no	W	16[3]		MFC_Tag	1	W	16 L
21	MFC Class ID and Command Opcode	MFC_Cmd	16	yes	W	32	x'03014'	MFC_ClassID_CMD	8	W	32
—	MFC Command Status[4]	—	—	—	—	—		MFC_CMDStatus	1	R	32
—	MFC Queue Status[5]	—	—	—	—	—		MFC_QStatus	1	R	32

Table 17-2. SPE Channels and Associated MMIO Registers (Page 3 of 3)

SPE Channel #	Name	Channel Interface[1] (MFC SPU Command Queue)					MMIO Register Interface[1] (MFC Proxy Command Queue)				
		Mnemonic	Max. Entries	Blocking	R/W	Width (bits)	Offset From Base	Mnemonic	Max. Entries	R/W	Width (bits)
MFC Tag Status											
22	MFC Write Tag-Group Query Mask	MFC_WrTagMask	1	no	W	32	x'0321C'	Prxy_QueryMask	1	R/W	32
23	MFC Write Tag Status Update Request[6]	MFC_WrTagUpdate	1	yes	W	32	—	—	—	—	—
—	Proxy Tag-Group Query-Type[6]	—	—	—	—	—	x'03204'	Prxy_QueryType	1	R/W	32
24	MFC Read Tag-Group Status	MFC_RdTagStat	1	yes	R	32	x'0322C'	Prxy_TagStatus	1	R	32
25	MFC Read List Stall-and-Notify Tag Status[7]	MFC_RdListStallStat	1	yes	R	32	—	—	—	—	—
26	MFC Write List Stall-and-Notify Tag Acknowledgement[7]	MFC_WrListStallAck	1	no	W	32	—	—	—	—	—
27	MFC Read Atomic Command Status[8]	MFC_RdAtomicStat	1	yes	R	32	—	—	—	—	—
SPU Mailboxes											
28	SPU Write Outbound Mailbox	SPU_WrOutMbox	1	yes	W	32	X'04004'	SPU_Out_Mbox	1	R	32
29	SPU Read Inbound Mailbox	SPU_RdInMbox	4	yes	R	32	X'0400C'	SPU_In_Mbox	4	W	32
30	SPU Write Outbound Interrupt Mailbox[9]	SPU_WrOutIntrMbox	1	yes	W	32	X'04000'	SPU_Out_Intr_Mbox	1	R	64
—	SPU Mailbox Status[10]	—	—	—	—	—	X'04014'	SPU_Mbox_Stat	1	R	32
31:127	Reserved[11]	—	—	—	—	—	—	—	—	—	—

1. Because the channel and MMIO command queues are separate queues, a MMIO register has a function that is similar to its corresponding channel, but the MMIO register does not affect the operation of that channel, which is a separate function.
2. On the MMIO register interface, writing the SPU_Cfg register configures the signal-notification to operate in either OR mode or overwrite mode. On the SPU channel interface, no such configuration function is available.
3. When setting a 16-bit channel using a *SPU C/C++ Language Extensions* intrinsic, only the least-significant 16 bits of a 32-bit scalar unsigned int operand is used.
4. On the MMIO register interface, reading the MFC_CMDStatus register returns information about the success or failure of queuing a DMA command. On the SPU channel interface, comparable information can be obtained by writing a command to the MFC_Cmd channel, which will block until a queue entry is available.
5. On the MMIO register interface, reading the MFC_QStatus register returns the number of queue entries available. On the SPU channel interface, comparable information can be obtained by issuing a channel-count (**rchcnt**) instruction.
6. The MFC_WrTagUpdate channel controls the circumstances under which the status in the MFC_RdTagStat channel is updated. The Prxy_QueryType MMIO register performs a comparable function.
7. Only an SPU can issue a DMA list command.
8. Only an SPU can issue MFC atomic commands.
9. Access to this MMIO register is available only to privileged PPE software.
10. On the MMIO register interface, reading the SPU_Mbox_Stat MMIO register returns the number of available mailbox entries. On the SPU channel interface, comparable information can be obtained by issuing a channel-count (**rchcnt**) instruction.
11. The *Synergistic Processor Unit Instruction Set Architecture* supports 128 channels, based on the 7-bit field in the channel command.

17.1.5 Channel Instructions

The SPU instruction set defines three instructions to access channels. *Table 17-3* lists the assembler instruction mnemonics and their corresponding C-language intrinsics. The assembler instructions are described below; the intrinsics are described in *Section B.2.3* on page 776.

Table 17-3. SPE Channel Instructions

Function	Assembler Instruction	C-Language Intrinsic	Description
Read Channel	**rdch** Rt, chnum	**d = spu_readch**(chnum) **d = spu_readchqw**(chnum)	Data from the channel is loaded into the GPR or C-language variable.
Write Channel	**wrch** chnum, Ra	**(void) spu_writech**(chnum, a) **(void) spu_writechqw**(chnum, a)	Data from the GPR or C-language variable is stored to the channel.
Read Channel Count	**rchcnt** Rt, chnum	**d = spu_readchcnt**(chnum)	The channel count is loaded into the GPR or C-language variable.

17.1.6 Channel Capacity and Blocking

For a non-blocking read channel, a read-channel instruction will execute without delay. Similarly, for a non-blocking write channel, a write-channel instruction will execute without delay. For a blocking channel, however, a read-channel or write-channel instruction can block and cause a delay (SPE stall) of arbitrary length. If a read-channel instruction is directed at a channel connected to an empty input queue (channel count is ‘0’), the SPE will stall until valid data appears in the queue or the SPE is interrupted. Similarly, if a write-channel instruction is directed at a channel connected to a full output queue (channel count is ‘0’), the SPE will stall until the queue is able to accept the write data or the SPE is interrupted.

An SPE uses channels to interact with producers and consumers of data that are inherently asynchronous to SPE software. Stalling on a channel read or write has the benefit of minimizing the power consumed by instruction execution, and it also reduces SPE software complexity. This may be useful for cases in which SPE software has no other work to perform until the channel capacity increases.

To avoid stalling on access to a blocking channel, SPE software can read the channel count to determine the available channel capacity; if the read-channel-count instruction returns ‘0’, a read- or write-channel instruction would stall.

Reading the channel count may be useful when SPE software has work to perform while waiting for space on the channel to become available (for channel writes) or data on the channel to become available (for channel reads). In this case, SPE software could be structured to intermittently check channel counts (polling) or the SPE interrupt facility can be enabled to implement true asynchronous event handling. Many of the channels have a corresponding event that can be enabled to cause an asynchronous interrupt, and each event interrupt can be enabled independently, as described in *Section 18 SPE Events* on page 463.

17.2 SPU Event-Management Channels

There are four SPU event-management channels:

- SPU Read Event Status (`SPU_RdEventStat`)—channel 0.

- SPU Write Event Mask (`SPU_WrEventMask`)—channel 1.
- SPU Write Event Acknowledgement (`SPU_WrEventAck`)—channel 2.
- SPU Read Event Mask (`SPU_RdEventMask`)—channel 11.

These are used to control and monitor event reporting. The bit assignments and definitions are the same for all four channels. In addition, there is an internal Pending Event register that is hidden from SPE software but visible to privileged PPE software.

For details about the SPU event-management channels, see *Section 18.2 Events and Event-Management Channels* on page 464.

17.3 SPU Signal-Notification Channels

There are two SPU signal-notification channels, one to read each of the two signal-notification registers. A signal is a short message from outside the SPU (that is, from the PPE, another SPE, or another system device) that can be from one to 32 bits long.

A device outside the SPU sends a signal-notification message to the SPU by writing to the main-storage address of an MMIO register in the SPU's MFC. The signal is latched in the MMIO register, and the SPU executes a read-channel (**`rdch`**) instruction to get the signal value. An SPU can send a signal-notification message to another SPU with its special send-signal instructions (for example, **`sndsig`**).

For details about the SPU signalling channels, including programming examples, see *Section 19.7 Signal Notification* on page 542.

17.4 SPU Decrementer

Each SPU contains a 32-bit decrementer implemented as a down-counter. All decrementers in the SPUs and PPE count down at the same rate (*Section 13.3.2 SPE Decrementers* on page 384). An SPU decrementer is accessed through two channels, the SPU Write Decrementer channel (`SPU_WrDec`) and the SPU Read Decrementer channel (`SPU_RdDec`).

A decrementer event becomes pending when the value in the decrementer changes from '0' to negative (the most-significant bit changes from '0' to '1'); see *Section 18 SPE Events* on page 463 for details. *Section 18.10.1* on page 497 gives examples of how a program can use an SPU decrementer event.

The SPU decrementer can run if the `Dh` (decrementer halt) bit in the privileged MFC Control Register (`MFC_CNTL`) is cleared to '0' by PPE software. To start a stopped decrementer, SPE software writes to the SPU Write Decrementer channel (`SPU_WrDec`). To stop a running decrementer, SPE software acknowledges the decrementer event when the decrementer event is disabled. The decrementer can also be stopped by setting the `MFC_CNTL[Dh]` bit to '1'. For additional details and code examples, see *Section 13.3.2 SPE Decrementers* on page 384.

17.4.1 SPU Write Decrementer Channel

The value written to the SPU Write Decrementer channel (`SPU_WrDec`) sets the starting value of the 32-bit decrementer. When SPE software writes `SPU_WrDec` with a write-channel (**`wrch`**) instruction, the decrementer starts running (counting down) if it was not already running. The value loaded into the decrementer determines the time lapse between the **`wrch`** instruction and the decrementer event becoming pending.

The decrementer event becomes pending when the most-significant bit of the decrementer changes from a ‘0’ to a ‘1’ (the value changes from ‘0’ to negative). If the value loaded into the decrementer causes a change from ‘0’ to ‘1’ in the most-significant bit, the event becomes pending immediately. Setting the decrementer to a value of ‘0’ results in an event after a single decrementer count.

The field assignments and definitions for SPU Write Decrementer are shown below.

Decrementer Count Value

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

Bits	Field Name	Description
0:31	Decrementer Count Value	Decrementer count value.

17.4.2 SPU Read Decrementer Channel

The value read from the SPU Read Decrementer channel (`SPU_RdDec`) is the current value of the 32-bit decrementer. Reading the decrementer count has no effect on the accuracy of the decrementer (a read does not affect the count-down rate).

The field assignments and definitions for SPU Read Decrementer are identical to those of the SPU Write Decrementer, shown above.

17.5 MFC Write Multisource Synchronization Request Channel

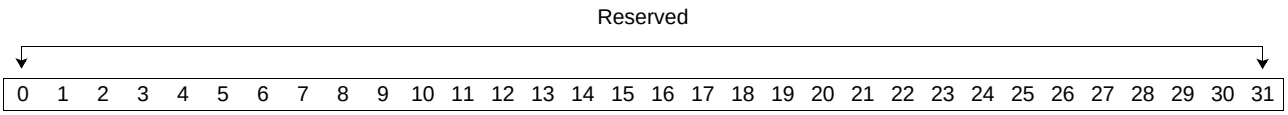
As described in *Section 20.1.5* on page 567, the CBE processor contains multiple address and communication domains, and the MFC Multisource Synchronization Facility must be used to ensure cumulative ordering across all address domains.

The value written to the MFC Write Multisource Synchronization Request channel (`MFC_WrMSSyncReq`) causes the MFC to start tracking outstanding write transfers to the MFC. The value written to `MFC_WrMSSyncReq` is ignored, but SPE software should write ‘0’ to ensure compatibility with future architectural revisions.

When software writes to `MFC_WrMSSyncReq`, the channel count is cleared to ‘0’ to indicate an outstanding request is active. When the synchronization is complete (all writes to the MFC that were started prior to the multi-source synchronization request are done), the channel count is set to ‘1’. When the channel count changes from ‘0’ to ‘1’, the multi-source synchronization event becomes pending (see *Section 18* on page 463).

When SPE software writes to this channel and the channel count is ‘0’, the SPU stalls until the write transfers being tracked by the currently active multi-source synchronization request are complete.

The field assignments and definitions for MFC Write Multisource Synchronization Request are shown below.



Bits	Field Name	Description
0:31	Reserved	Reserved.

See *Section 20.1.5 MFC Multisource Synchronization Facility* on page 567 for a description of multisource synchronization. Graphic flowcharts of the possible software sequences are given in *Figure 20-2* on page 569, *Figure 20-3* on page 571, and *Figure 20-4* on page 572.

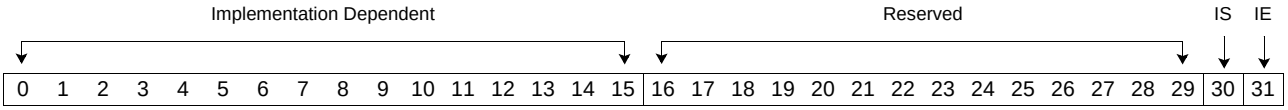
Section 18.6.2 Procedure for Handling the Multisource Synchronization Event on page 474 described details for handling a multisource synchronization event.

17.6 SPU Read Machine Status Channel

The value read from the SPU Read Machine Status channel (SPU_RdMachStat) contains two status bits: the isolation status (`IS`) and the SPU interrupt-enable status (`IE`). If the `IS` bit is set to ‘1’, the SPU is operating in the isolated state; if the `IS` bit is ‘0’, the SPU is not isolated.

If the `IE` bit is set to ‘1’, any SPU event that is both enabled and pending will cause the SPU to take an interrupt. If the `IE` bit is ‘0’, the SPU will not take an interrupt regardless of event status. SPE software can set and clear the `IE` bit by executing an indirect branch with the `E` feature bit or the `D` feature bit set, respectively. For more information, see *Section 18* on page 463.

The field assignments and definitions for SPU Read Machine Status are shown below.



Bits	Field Name	Description
0:15	Implementation Dependent	
16:29	Reserved	Cleared to ‘0’s.
30	IS	Isolation status. ‘0’ not isolated. ‘1’ isolated.
31	IE	SPU interrupt enable status. ‘0’ interrupts disabled. ‘1’ interrupts enabled.

17.7 SPU Write State Save-and-Restore Channel

The value written to the SPU Write State Save-and-Restore channel (`SPU_WrSRR0`) sets the return address used by the interrupt-return (**`iret`**) instruction. The most common reason to write `SPU_WrSRR0` is to restore interrupt state when nested interrupts are supported by software. For more information on interrupts and nested interrupts, see *Section 18* on page 463 and *Section 9 PPE Interrupts* on page 237.

SPE software should not write to this channel when interrupts are enabled; doing so can result in the contents of SRR0 being indeterminate. After software writes to this channel and before it executes any instruction that depends on the SRR0 value (such as an **`iret`** instruction), software must execute the **`syncc`** instruction (**`sync`** with the channel feature bit set).

The field assignments and definitions for SPU Write State Save-and-Restore are shown below.

SRR0 Data

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

Bits	Field Name	Description
0:31	SRR0 Data	State save/restore register 0 data.

17.8 SPU Read State Save-and-Restore Channel

The value read from the SPU Read State Save-and-Restore channel (`SPU_RdSRR0`) is the current contents of the SRR0 register, which is the return address saved when the SPU takes an interrupt. The most common reason to read `SPU_RdSRR0` is to save interrupt state when nested interrupts are supported by software. For more information on interrupts and nested interrupts, see *Section 18* on page 463 and *Section 9 PPE Interrupts* on page 237.

The field assignments and definitions for SPU Read State Save-and-Restore are identical to those for SPU Write State Save-and-Restore, shown above.

17.9 MFC Command Parameter Channels

The channels used by SPE software to enqueue an MFC command to the MFC SPU command queue are described in the following sections. There are six channel involved in setting up an MFC command as shown in *Table 17-4*.

Table 17-4. MFC Command Parameter and Command Opcode Channels (Page 1 of 2)

Channel Mnemonic	Channel Number	Description
`MFC_LSA`	16	MFC Local Storage Address
`MFC_EAH`	17	MFC Effective Address High
`MFC_EAL`	18	MFC Effective Address Low or List Address

Table 17-4. MFC Command Parameter and Command Opcode Channels (Page 2 of 2)

Channel Mnemonic	Channel Number	Description
`MFC_Size`	19	MFC Transfer Size or List Size
`MFC_TagID`	20	MFC Command Tag Identification
`MFC_Cmd`	21	MFC Command Opcode and MFC Class ID

The first five channels listed above are referred to as MFC command parameter channels; these channels determine the characteristics of an MFC command. If the SPU MFC command queue has capacity to accept the command, it is enqueued when the command opcode and class ID are written to MFC Command Opcode and MFC Class ID, channel 21. The MFC Command Opcode and MFC Class ID logical channels are the low and high half-words, respectively, of a single 32-bit value that is written to channel 21 to complete the command-construction sequence.

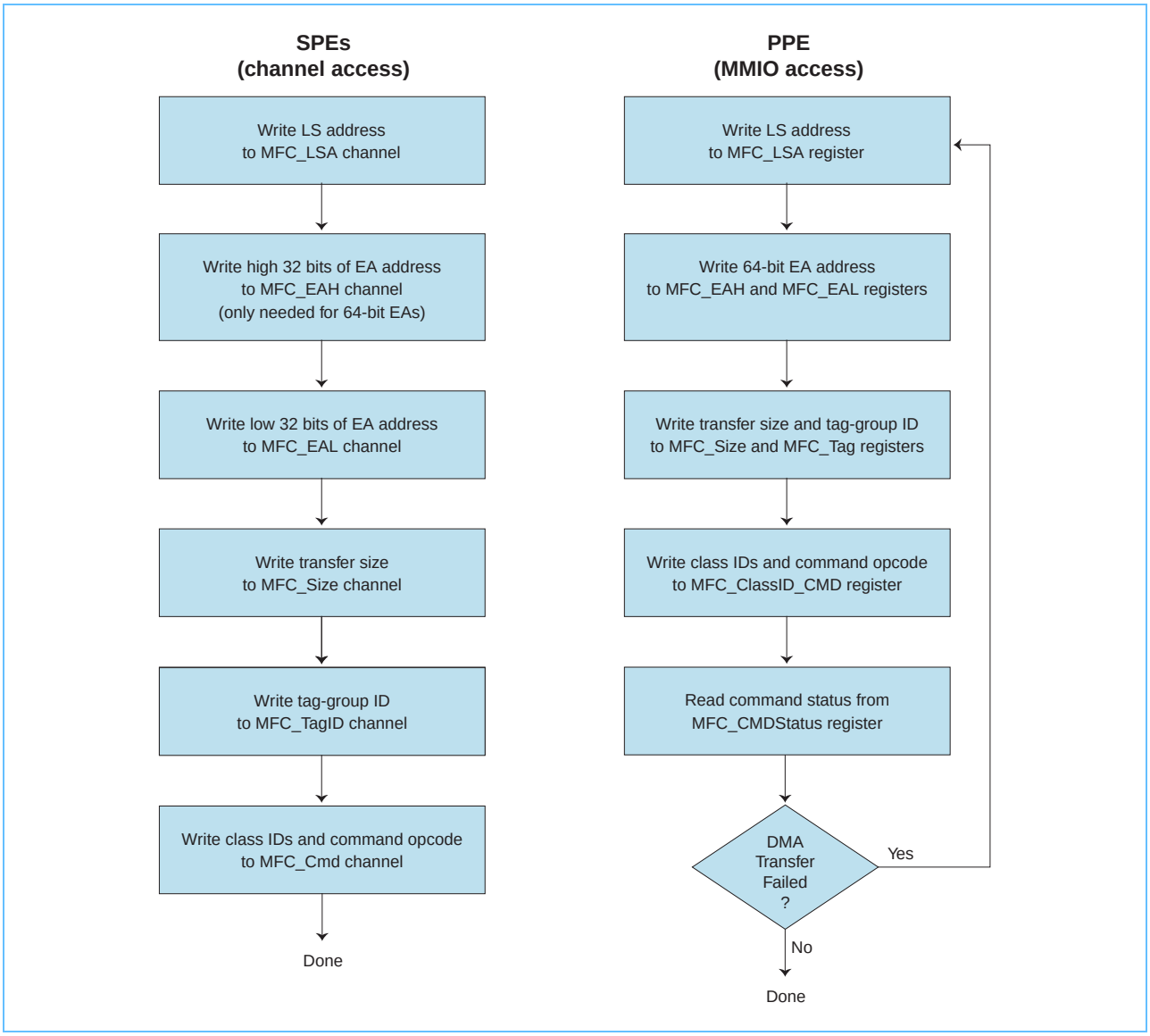
Whenever SPE software writes to MFC Command Opcode, the MFC attempts to enqueue a complete MFC command. Thus, the parameter channels must be written before MFC Command Opcode is written. The only exception is MFC Effective Address High, which need not be written if the default value of ‘0’ is acceptable. SPE software must set all other command parameters before it writes MFC Command Opcode; otherwise, improper operation of the MFC command queue may result.

The parameter channels can be written in any order so long as MFC Command Opcode and MFC Class ID is written last, and if a parameter channel is written more than once, the last value written is used in the queued command.

The MFC command parameter channels (the first five listed above) are non-blocking and do not have channel counts associated with them. Performing a read channel count (**`rchcnt`**) instruction on any of these channels returns ‘1’. The MFC Command Opcode channel is blocking and has a maximum channel count of 16. Performing a **`rchcnt`** instruction on this channel returns the remaining capacity (number of empty queue entries) in the MFC command queue.

Figure 17-2 shows the sequences in which MFC commands are issued by SPE software and by PPE software. For additional details about the MFC commands, including DMA-command programming steps and example code, see *Section 19.2 MFC Commands* on page 506.

Figure 17-2. Sequences for Issuing MFC Commands



17.9.1 MFC Local Storage Address Channel

The value written to the MFC Local Storage Address channel (`MFC_LSA`) sets the SPU LS address (LSA) for the MFC command being formed. This address is used as the source (for a **`put`** operation) or destination (for a **`get`** operation) of the MFC transfer as defined in the MFC command. For more information, see *Section 19* on page 505.

If the LSA is unaligned, MFC command queue processing is suspended, and an MFC DMA alignment interrupt is raised to the PPE. To be considered aligned, the four least significant bits of the LS address must match the least-significant four bits of the effective address (MFC Effective Address Low or List Address Channel (see page 453)).

For best performance on transfers of 128 or more bytes, the MFC LSA and the MFC Effective Address Low (EAL) should have bits 25 through 31 cleared to ‘0’, that is, the address should be aligned on a 128-byte boundary.

The field assignments and definitions for MFC LSA are shown below.

MFC Local Storage Address

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

Bits	Field Name	Description
0:31	MFC Local Storage Address	MFC local storage address.

17.9.2 MFC Effective Address High Channel

The value written to the MFC Effective Address High channel (`MFC_EAH`) sets the most-significant 32 bits of the 64-bit effective address for the MFC command being formed. If MFC translation is enabled by the PPE (Relocate bit in the MFC state register is set to ‘1’), the MFC translates effective addresses into real addresses as described in *Book III* of the *PowerPC Architecture*.

MFC Effective Address High defaults to ‘0’; this is the only MFC command parameter channel that SPE software need not write before it writes to the MFC Command Opcode channel. If software does not write MFC Effective Address High, the effective address of the queued MFC command will be in the lowest 4 GB of the effective address space.

If the address is invalid due to a segment fault, a mapping fault, or other address violation, MFC command queue processing is suspended, and an interrupt is raised to the PPE.

The MFC checks the validity of the effective address during transfers. Partial transfers can be performed before the MFC encounters an invalid address and raises the interrupt to the PPE.

The field assignments and definitions for MFC Effective Address High are shown below.

High Word of 64-bit Effective Address (Optional)

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

Bits	Field Name	Description
0:31	High Word of 64-bit Effective Address	High word of the 64-bit effective address.

17.9.3 MFC Effective Address Low or List Address Channel

The value written to the MFC Effective Address Low or List Address channel (`MFC_EAL`) sets either the least-significant 32 bits of the 64-bit effective address for the MFC command being formed or the LS address of the list of list elements for the MFC DMA list command being formed. If MFC translation is enabled by the PPE (Relocate bit in the MFC state register is set to ‘1’), the MFC translates effective addresses into real addresses as described in *Book III* of the *PowerPC Architecture*.

For transfer sizes less than 16 bytes, the MFC Effective Address Low must be naturally aligned (bits 28 through 31 must provide natural alignment based on the transfer size). For transfer sizes of 16 bytes or greater, the MFC Effective Address Low must be aligned to at least a 16-byte boundary (bits 28 through 31 must be ‘0’).

For best performance on transfers of 128 or more bytes, the MFC Effective Address Low (and the MFC Local Storage Address Channel (see page 452)) should have bits 25 through 31 cleared to ‘0’, that is, the address should be aligned on a 128-byte boundary.

For MFC list commands, the MFC List Address must be aligned on an eight-byte boundary (bits 29 through 31 of the List Address must be ‘0’).

If the address is invalid due to a segment fault, a mapping fault, or other address violation, MFC command queue processing is suspended, and an interrupt is raised to the PPE. The MFC checks the validity of the effective address during transfers. Partial transfers can be performed before the MFC encounters an invalid address and raises the interrupt to the PPE.

The field assignments and definitions for MFC Effective Address Low or List Address are shown below.

Low Word of 64-bit Effective Address or the Local Storage Address of the MFC List

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

Bits	Field Name	Description
0:31	Low Word of 64-bit Effective Address, or the Local Storage Address of the MFC List	Low word of the 64-bit effective address or the LS address of the MFC list.

17.9.4 MFC Transfer Size or List Size Channel

The value written to the MFC Transfer Size or List Size channel (`MFC_Size`) sets either the size of the MFC transfer for the command being formed or the size of the list for an MFC DMA list command being formed. In both cases, the value written specifies the number of bytes and cannot be larger than 16 KB.

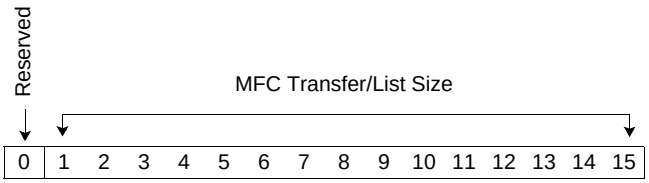
The transfer size can have a value of 1, 2, 4, 8, 16, or a multiple of 16 bytes to a maximum of 16 KB. The source and destination addresses (MFC Local Storage Address Channel (see page 452) and MFC Effective Address Low or List Address Channel (see page 453)) must be naturally aligned to the transfer size for sizes up to 16 bytes. For sizes greater than 16 bytes, alignment on a 16-byte boundary is sufficient.

The List Size of an MFC DMA list command can range from eight bytes up to a maximum of 16 KB. Since each list element has a size of eight bytes, the list can be from one to 2048 elements long. The address of the list (MFC Effective Address Low or List Address Channel (see page 453)) in LS must be aligned on an eight-byte boundary.

If the size is invalid, MFC command queue processing is suspended and an MFC DMA alignment interrupt is raised to the PPE.

Transfers of less than one cache line (128 bytes) should be used sparingly; excessive use of short transfers wastes bus and memory bandwidth. Transfers of less than 16 bytes should only be used to communicate with an I/O device since the setup cost of the transfer is high and bus bandwidth is wasted. When the transfer size is 128 bytes or longer, programmers should align the source and destination addresses on 128-byte boundaries.

The field assignments and definitions for MFC Transfer Size or List Size are shown below.



Bits	Field Name	Description
0	Reserved	Cleared to ‘0’
1:15	MFC Transfer/List Size	MFC transfer size or list size

17.9.5 MFC Command Tag Identification Channel

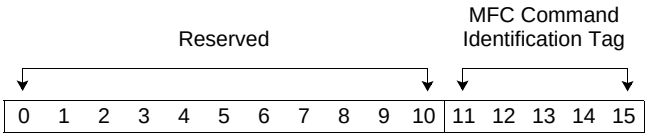
The value written to the MFC Command Tag Identification channel (`MFC_TagID`) sets the tag value for the command being formed. The identification tag can be any value between 0 and 31. Identification tags have a purely local scope in the hardware.

When two or more MFC commands are enqueued with the same tag value, the commands are in the same tag group. Any number of MFC commands can be tagged with the same identification value.

Tag groups can be formed only within a single MFC command queue. Thus, tags assigned to commands in the MFC SPU command queue are independent of the tags assigned to commands in the MFC proxy command queue.

If the Reserved field (bits 0 through 10) is not cleared to ‘0’, MFC command queue processing is suspended, and an interrupt is raised to the PPE.

The field assignments and definitions for MFC Command Tag Identification are shown below.



Bits	Field Name	Description
0:10	Reserved	Cleared to ‘0’s.
11:15	MFC Command Tag Identification	MFC command tag identification.

17.9.6 MFC Class ID and MFC Command Opcode Channel

The value written to the MFC Class ID and MFC Command Opcode channel sets the Replacement Class ID (`RclassID`), the Transfer Class ID (`TclassID`), and the command opcode for the MFC command that has been formed by previous writes to the MFC command parameter chan-

nels (see MFC Command Parameter Channels on page 450). In addition, the write to this channel has the side-effect of enqueuing the complete MFC command into the SPU MFC command queue. The write to this channel must be the last channel write during MFC command formation.

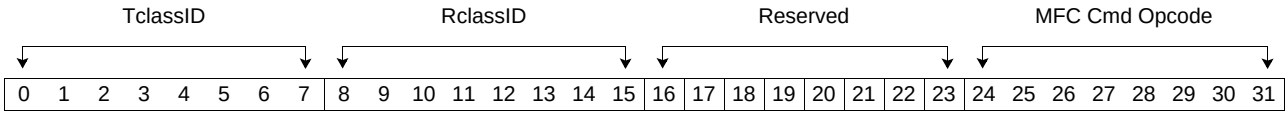
The `RclassID` and `TclassID` are used by the processor and software to improve the overall performance and throughput of the system. The `RclassID` influences L2-cache and TLB replacement, as described in *Section 6.3 Replacement Management Tables* on page 154. The `TclassID` influences the allocation of bus bandwidth, as described in *Section 19.2.7 Replacement Class ID and Transfer Class ID* on page 513.

The MFC Command Opcode determines the operation for the MFC command being formed. If the MFC command opcode or any of the command parameters is invalid, MFC command-queue processing is suspended and an invalid MFC command interrupt is raised to the PPE. See *Section 19* on page 505 for a complete discussion of MFC commands.

Software must avoid placing commands in the queue with forward dependencies on newer commands placed in the queue. A string of commands with this type of dependency can create a deadlock, depending on the number of available slots in the MFC queue. Queue depth depends on the CBE implementation version, so software must not be written to require a specific queue depth.

Software can read the channel count of `MFC_Cmd` to determine the number of available queue entries. Since a write to `MFC_Cmd` will stall when the MFC queue is full, software can use the channel count to avoid stalling by waiting for the channel count to become greater than ‘0’.

The field assignments and definitions for MFC Class ID and MFC Command Opcode are shown below.



Bits	Field Name	Description
0:7	TclassID	Transfer class identifier.
8:15	RclassID	Replacement class identifier.
16:23	Reserved	Should normally be cleared to ‘0’s. Bit 16 set to ‘1’ indicates that the opcode is reserved.
24:31	MFC Cmd Opcode	MFC command opcode.

17.10 MFC Tag-Group Management Channels

When MFC commands are entered into the command queue, each command is tagged with a 5-bit tag-group identifier, the MFC command tag identifier. The same identifier can be used for multiple MFC commands to create a tag group containing all the commands currently in the queue with the same command tag. Software can use the MFC command tag to check the completion of all queued commands in a tag group. In addition, the MFC command tag is used by hardware to inform software that an MFC DMA list command has reached an element with its stall-and-notify flag set. Software then uses the command tag to cause the MFC to resume the stalled MFC DMA list command. For more information, see *Section 19* on page 505.

17.10.1 MFC Write Tag-Group Query Mask Channel

The value written to the MFC Write Tag-Group Query Mask channel (`MFC_WrTagMask`) selects the tag groups to be included in a query operation. A query (MFC tag status update request) operation is started by writing to the MFC Write Tag Status Update Request channel; the status of the query is available in the MFC Read Tag-Group Status channel.

The value written to this channel is retained by the MFC until changed by a subsequent write. Therefore, the same value can be used for multiple status queries. If this value is changed by software when a query is pending, the results of the query are ambiguous. A pending query should always be cancelled before a modification of this mask. Software cancels a query by writing ‘0’ to the MFC Write Tag Status Update Request channel.

Software can get the current value in this channel by reading the MFC Read Tag-Group Query Mask channel.

The field assignments and definitions for MFC Write Tag-Group Query Mask are shown below.

g_{1F}	g_{1E}	g_{1D}	g_{1C}	g_{1B}	g_{1A}	g_{19}	g_{18}	g_{17}	g_{16}	g_{15}	g_{14}	g_{13}	g_{12}	g_{11}	g_{10}	g_{F}	g_{E}	g_{D}	g_{C}	g_{B}	g_{A}	g_{9}	g_{8}	g_{7}	g_{6}	g_{5}	g_{4}	g_{3}	g_{2}	g_{1}	g_{0}
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

Bits	Field Name	Description
0:31	g_n	Tag group “n” select. 0 Tag group not in query. 1 Tag group is in query.

17.10.2 MFC Read Tag-Group Query Mask Channel

The value read from the MFC Read Tag-Group Query Mask channel (`MFC_RdTagMask`) is the most recent value written to the MFC Write Tag-Group Query Mask channel (`MFC_WrTagMask`). Software can read this channel during SPU context save and restore operations and as an alternative to keeping shadow copies of the `MFC_WrTagMask` channel.

The field assignments and definitions for MFC Read Tag-Group Query Mask are identical to those for MFC Write Tag-Group Query Mask, shown above.

17.10.3 MFC Write Tag Status Update Request Channel

The value written to the MFC Write Tag Status Update Request channel (`MFC_WrTagUpdate`) controls when the MFC tag-group status is updated in MFC Read Tag-Group Status. The MFC Write Tag-Group Query Mask channel controls which tag groups participate in the update request and therefore which tag groups influence the value read from MFC Read Tag-Group Status.

The value written to MFC Write Tag Status Update determines when MFC Read Tag-Group Status is updated; three update options are possible:

- Status updated immediately after `MFC_WrTagUpdate` is written,
- Status updated when any enabled tag group completes after `MFC_WrTagUpdate` is written,
- Status updated when all enabled tag groups complete after `MFC_WrTagUpdate` is written.

Software must write MFC Write Tag Status Update Request before it reads MFC Read Tag-Group Status.

An MFC Write Tag Status Update request (writing to `MFC_WrTagUpdate`) should be issued after writing a value to MFC Write Tag-Group Query Mask and after enqueuing the MFC commands for the tag groups of interest. If the commands for a tag group are completed before issuing the MFC Write Tag Status Update request thereby satisfying the update status condition, the status is returned without waiting.

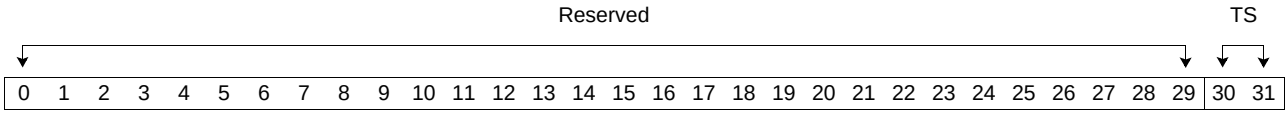
If software reads from MFC Read Tag-Group Status without first writing to MFC Write Tag Status Update Request, this results in a software-induced deadlock.

If software writes to MFC Tag Status Update Request and later needs to cancel the status update request, software should execute the following steps:

1. Request an immediate update status by writing ‘0’ to `MFC_WrTagUpdate`.
2. Read the channel count for `MFC_WrTagUpdate` until a value of ‘1’ is returned.
3. Read from the MFC Read Tag Group Status to discard the unwanted query result and reset the channel count for MFC Read Tag Group Status to ‘0’.

If software issues two conditional update requests (executes to writes to `MFC_WrTagUpdate`) without an intervening read of the MFC Read Tag Group Status channel, the value of MFC Read Tag Group Status is unpredictable. To avoid the unpredictable status, software should pair every write to MFC Write Status Update Request with a read from MFC Read Tag-Group Status. The exception is when software cancels a request using the steps shown above.

The field assignments and definitions for MFC Write Tag Status Update Request are shown below.



Bits	Field Name	Description
0:29	Reserved	Reserved.
30:31	TS	Tag-status update condition: 00 MFC_TAG_UPDATE_IMMEDIATE: Update tag status immediately, unconditional. 01 MFC_TAG_UPDATE_ANY: Update tag status if or when any enabled tag group completes. 10 MFC_TAG_UPDATE_ALL: Update tag status if or when all enabled tag groups complete. 11 Reserved.

17.10.4 MFC Read Tag-Group Status Channel

The value read from the MFC Read Tag-Group Status channel (`MFC_RdTagStat`) reports the status of the tag groups from the last tag-group status update request. Software requests a tag-group status update by writing to MFC Write Tag Status Update Request, and the status reported by MFC Read Tag-Group Status includes only those tag groups enabled by the value that software writes to the MFC Write Tag-Group Query Mask channel. The bits in the status that correspond to tag groups disabled by the value in MFC Write Tag-Group Query Mask will be ‘0’.

Software must write a request value to MFC Write Tag Status Update Request before reading from MFC Read Tag-Group Status; failure to do so results in a software-induced deadlock, and only privileged software on the PPE can remove the deadlock condition.

The field assignments and definitions for MFC Read Tag-Group Status are shown below.

g_{1F}	g_{1E}	g_{1D}	g_{1C}	g_{1B}	g_{1A}	g_{19}	g_{18}	g_{17}	g_{16}	g_{15}	g_{14}	g_{13}	g_{12}	g_{11}	g_{10}	g_{F}	g_{E}	g_{D}	g_{C}	g_{B}	g_{A}	g_{9}	g_{8}	g_{7}	g_{6}	g_{5}	g_{4}	g_{3}	g_{2}	g_{1}	g_{0}
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

Bits	Field Name	Description
0:31	g_n	Tag group “n” status. 0 Tag group not complete. 1 Tag group is complete.

17.10.5 MFC Read List Stall-and-Notify Tag Status Channel

The value read from MFC Read List Stall-and-Notify Tag Status channel (`MFC_RdListStallStat`) reports which tag groups have an MFC DMA list command in the stall state due to a list element with the stall-and-notify flag set to ‘1’.

List elements for an MFC list command contain a transfer size, the low 32-bits of an effective address, and a stall-and-notify flag. If the flag is set on a list element, the MFC completes the transfer specified by the list element then stops executing the MFC list command and sets the bit in `MFC_RdListStallStat` corresponding to the tag group of the MFC list command. An MFC list command remains stalled until acknowledged by writing the tag value to the MFC Write List Stall-and-Notify Tag Acknowledgment.

An application program can use the DMA list stall-and-notify capability to monitor the progress of a DMA list command. It can also use stall-and-notify capability when it needs to modify the characteristics (transfer sizes and/or effective addresses) of list elements that follow the stalled list element. If an application determines that unprocessed list elements should be skipped, it can set the transfer size to ‘0’ in all the elements to be skipped. Software can cancel elements this way because MFC DMA hardware is not allowed to prefetch list elements beyond an element that has its stall-and-notify flag set to ‘1’.

When software reads MFC Read List Stall-and-Notify Tag Status, all the bits in the status are reset to ‘0’ and the channel count is cleared to ‘0’. Between reads of the MFC Read List Stall-and-Notify Tag Status, the status accumulates the stall-and-notify status of tag groups as they occur; the channel count for MFC Read List Stall-and-Notify tag Status never increments above ‘1’.

If software reads from MFC Read List Stall-and-Notify Tag Status when the channel count is ‘0’, the SPU will stall until the MFC encounters a list element with the stall-and-notify flag set to ‘1’. Thus, software must not read from MFC Read List Stall-and-Notify Tag Status when the channel count is ‘0’ (no bits set in the status) and no outstanding list elements for commands in the MFC queue have the stall-and-notify flag set to ‘1’; if software violates this rule, a software-induced deadlock will occur with the SPU stalled on the read of MFC Read List Stall-and-Notify Tag Status.

Software must also use DMA list commands with the fence or barrier feature bit set if one tag group contains multiple DMA list commands with elements that have a stall-and-notify flag set to ‘1’. Without a fence or barrier to enforce ordering, MFC DMA hardware is free to execute MFC commands (but not list elements) out of order. Thus, without enforced ordering, software cannot be sure which element with the stall-and-notify flag set to ‘1’ has been reached. Software must also implement some mechanism (for example, a linked list) to track DMA list commands and list elements as stalls occur.

The field assignments and definitions for MFC Read List Stall-and-Notify Tag Status are shown below.

g_{1F}	g_{1E}	g_{1D}	g_{1C}	g_{1B}	g_{1A}	g_{19}	g_{18}	g_{17}	g_{16}	g_{15}	g_{14}	g_{13}	g_{12}	g_{11}	g_{10}	g_{F}	g_{E}	g_{D}	g_{C}	g_{B}	g_{A}	g_{9}	g_{8}	g_{7}	g_{6}	g_{5}	g_{4}	g_{3}	g_{2}	g_{1}	g_{0}
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

Bits	Field Name	Description
0:31	g_n	Tag group “n” status. 0 Tag group not stalled. 1 Tag group stalled.

17.10.6 MFC Write List Stall-and-Notify Tag Acknowledgment Channel

The value written to the MFC Write List Stall-and-Notify Tag Acknowledgment channel (`MFC_WrListStallAck`) is the tag group number for a tag group with a stalled DMA list command; writing this tag value to `MFC_WrListStallAck` restarts (acknowledges) the stalled DMA list command in the tag group.

In contrast to most of the other tag group management channels, which have one bit per tag group, MFC Write List Stall-and-Notify Tag Acknowledgement accepts a value corresponding to a tag group number. A tag group number is computed as (31 – tag-group-bit-position), where tag-group-bit-position is the tag group’s bit position in the MFC Read Tag Group Status. MFC Write List Stall-and-Notify Tag Acknowledgement accepts only one tag group number per write, so each write to this channel can restart only one stalled tag group.

If software writes to this channel to acknowledge a tag group that is not stalled due to a stall-and-notify condition, MFC Read List Stall-and-Notify Tag Status will contain an invalid status.

The field assignments and definitions for MFC Write List Stall-and-Notify Tag Acknowledgement are shown below.

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

Bits	Field Name	Description
0:26	Reserved	Reserved.
27:31	MFC Tag	Tag ID of stalled group that is to be restarted.

17.11 MFC Read Atomic Command Status Channel

The value read from the MFC Read Atomic Command Status channel (`MFC_RdAtomicStat`) reports the status of the most-recently completed immediate MFC atomic-update command; immediate MFC atomic-update commands are **`getllar`**, **`putllc`**, or **`putlluc`**. This channel does not report any status for the queued **`putqlluc`** MFC atomic-update command. When software reads this channel, its contents are reset to ‘0’.

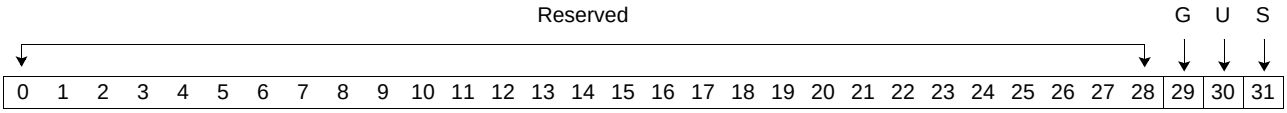
Reading MFC Read Atomic Command Status before issuing one of the immediate MFC atomic-update commands results in a software-induced deadlock.

Software should always read MFC Read Atomic Command Status after every immediate MFC atomic-update command; a read should be paired with each command. If software issues two or more immediate MFC atomic-update commands without intervening reads of the MFC Read Atomic Command Status channel, the channel will return an incorrect status.

Software can read the channel count for MFC Read Atomic Command Status to determine if the most-recently issued immediate MFC atomic-update command has completed. If the channel count is ‘0’, the command has not completed; if the channel count is ‘1’, the command has completed.

Completion of a subsequent immediate MFC atomic update command overwrites the status of earlier MFC commands.

The field assignments and definitions for MFC Read Atomic Command Status are shown below.



Bits	Field Name	Description
0:28	Reserved	Reserved.
29	G	Set if the get lock-line and reserve (**getllar**) command completed.
30	U	Set if the put lock-line unconditional (**putlluc**) command completed.
31	S	Put lock-line conditional command (**putllc**). 1 Put conditional unsuccessful. The reservation was lost. 0 Put conditional successful.

17.12 SPU Mailbox Channels

Mailboxes support the sending and buffering of 32-bit messages between an SPE and other devices, such as the PPE and other SPEs. Each SPE can access three mailbox channels, each of which is connected to a mailbox queue in the SPU’s MFC. Two one-entry mailbox channels—the SPU Write Outbound Mailbox and the SPU Write Outbound Interrupt Mailbox—are provided for sending messages from the SPE to the PPE or other device. One four-entry mailbox channel—the SPU Read Inbound Mailbox—is provided for sending messages from the PPE, or other SPEs or devices, to the SPE. Each of the two outbound mailbox channels has a corresponding MMIO register.

An SPE sends a mailbox message by writing the 32-bit message value to either its two outbound mailbox channels. The PPE and other devices can read a message in an outbound mailbox by reading the MMIO register in the SPE's MFC that is associated with the mailbox channel. Likewise, the PPE and other devices send messages to the inbound mailbox by writing the associated MMIO register. For interrupts associated with the SPU Write Outbound Interrupt Mailbox, there is no ordering of the interrupt and previously issued MFC commands.

For details about the mailbox channels, including programming examples, see *Section 19.6 Mailboxes* on page 531.

18. SPE Events

This chapter describes how SPE software can enable and process external events. From the perspective of SPE software, an event is generally an asynchronous change in the state of an resource or unit in the CBE processor that is external to that SPE. Each event of potential interest to an SPE has a corresponding bit in the SPE's Event Status channel. For example, the arrival of a mailbox message from another processor element sets the mailbox bit in the event status and may generate an SPU interrupt. SPE software can choose to monitor and respond to events synchronously (by polling or stalling) or asynchronously (by enabling the event interrupt).

Each SPE supports an event facility with the capability to mask and unmask events, wait on events, poll for events, and generate interrupts to the SPE when a specific event occurs. If the interrupts are enabled, an occurrence of an unmasked event results in the SPE's interrupt handler being invoked. These interrupts are referred to as "SPU interrupts"; they are unrelated to the PPE interrupts described in *Section 9 PPE Interrupts* on page 237.

18.1 Introduction

The main interface between SPE software and the event-tracking hardware is provided by four channels that control event handling and report event status. SPE software deals with events using four basic actions involving these three channels:

- Initialize event handling: write to Event Mask, channel 1.
- Recognize event(s) that have occurred: read Event Status, channel 0.
- Clear event(s): write Event Acknowledge, channel 2.
- Service event(s): execute application-specific code.

For a pictorial view of the manner in which events occur, see the Logical Representation of SPU Event Support figure in the *Cell Broadband Engine Architecture* specification. See *Section 17.2 SPU Event-Management Channels* on page 447 for details about the channels.

Typically, an SPE program is interested in only some of the possible SPE events. To enable only the events that are relevant to its operation, SPE software initializes a mask value with event bits set for the relevant events. After software recognizes an event, it writes an acknowledgement value so that the same events can be recognized again.

SPE software can choose to recognize events synchronously or asynchronously. For synchronous event handling, software can either poll for pending events (for example, test a channel count in a loop) or block when no enabled events are pending (for example, stall on a read from an empty channel). For asynchronous event handling, software must enable the event interrupt and provide interrupt handling code. An intermediate approach is to sprinkle Branch Indirect and Set Link if External Data (**bisled**) instructions, either manually or automatically using code-generation tools, throughout application code so that they are executed frequently enough to approximate asynchronous event detection (see *Section 18.5.1* on page 470). This approach has the advantage that interrupts need not be disabled and re-enabled around critical sections of code.

18.2 Events and Event-Management Channels

Each SPE can enable and respond to twelve different external events, listed in *Table 18-2* on page 464. There are four event-management channels, with identical bit definitions, that deal with masking, enabling, and reporting events:

- SPU Read Event Status (`SPU_RdEventStat`)—channel 0.
- SPU Write Event Mask (`SPU_WrEventMask`)—channel 1.
- SPU Write Event Acknowledgement (`SPU_WrEventAck`)—channel 2.
- SPU Read Event Mask (`SPU_RdEventMask`)—channel 11.

In addition to these four channels, there is an internal Pending Event register that is hidden from SPE software but visible to privileged PPE software. The internal register and the four event-management channels are described below.

18.2.1 Event Conditions and Bit Definitions for Event-Management Channels

The four event-management channels, plus the internal Pending Event register, use 32-bit values that defined as shown in *Table 18-2* on page 464. This table also explains the condition that leads to each event becoming pending. All of these events that can cause SPU interrupts, as described in *Section 18.3 SPU Interrupt Facility* on page 468.

For more information on conditions that lead to events being raised and the proper protocol for acknowledging and handling events, see *Section 18.6 Event-Specific Handling Guidelines* on page 472.

Table 18-1. Bit Assignments for Event-Management Channels and Internal Register

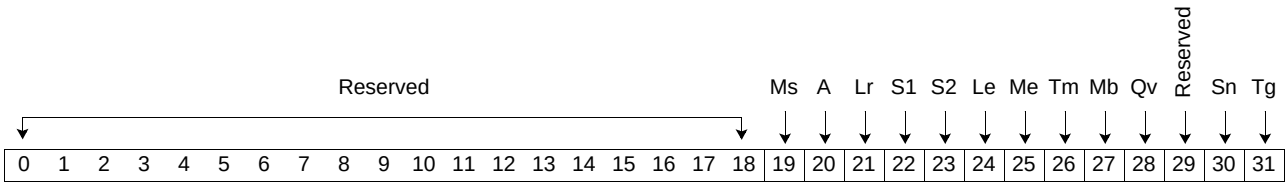


Table 18-2. Bit Definitions for Event-Management Channels and Internal Register (Page 1 of 2)

Bit	Field Name	Event Name	Condition for Event To Become Pending	Reason for Condition	Reference
0:18	—	Reserved			
19	Ms	Multisource Sync	Channel count for `MFC_WrMSSyncReq` changes from ‘0’ to ‘1’.	All of the write transfers directed at the SPE’s MFC that were started prior to the multisource synchronization request (the write to `MFC_WrMSSyncReq`) have completed.	*Section 18.6.2* on page 474
20	A	Privileged attention	PPE software or other device sets the attention event request bit to ‘1’ in the SPU Privileged Control Register (`SPU_PrivCntl`).	This can be done to control the execution of the SPU. When it is done, a Privileged Attention Event is raised on the SPU. The SPU can reset the condition by acknowledging the event.	*Section 18.6.3* on page 476

Table 18-2. Bit Definitions for Event-Management Channels and Internal Register (Page 2 of 2)

Bit	Field Name	Event Name	Condition for Event To Become Pending	Reason for Condition	Reference
21	Lr	Lock-line reservation-lost	A lock-line reservation created by a **getllar** MFC command is lost.	The MFC snooped a write to data in the reserved lock-line.	*Section 18.6.4* on page 476
22	S1	Signal-notification 1	Channel count for `SPU_RdSigNotify1` changes from '0' to '1'.	Unread signals are pending.	*Section 18.6.5* on page 477
23	S2	Signal-notification 2	Channel count for `SPU_RdSigNotify2` changes from '0' to '1'.		*Section 18.6.6* on page 478
24	Le	SPU Write Outbound Mailbox available	Channel count for `SPU_WrOutMbox` changes from '0' to non-zero.	A read from a full SPU Outbound (Interrupt) Mailbox register has occurred, and the SPU Write Outbound (Interrupt) Mailbox channel can be written without stalling the SPU.	*Section 18.6.7* on page 479
25	Me	SPU Write Outbound Interrupt Mailbox available	Channel count for `SPU_WrOutIntrMbox` changes from '0' to non-zero.		*Section 18.6.8* on page 480
26	Tm	Decrementer	The most-significant bit of the decrementer count changes from '0' to '1' (count changes from '0' to minus-'1').	The decrementer's elapsed time has expired. If software loads a value that causes the MSb to change from '0' to '1', the event becomes pending immediately.	*Section 18.6.9* on page 480
27	Mb	SPU Read Inbound Mailbox Available	Channel count for `SPU_RdInMbox` changes from '0' to non-zero.	A write to an empty SPU Read Inbound Mailbox register has occurred, and the SPU Read Inbound Mailbox channel can be read without stalling the SPU.	*Section 18.6.10* on page 482
28	Qv	MFC SPU Command Queue Available	Channel count for `MFC_Cmd` changes from '0' non-zero.	An entry in the MFC SPU Command Queue has become available.	*Section 18.6.11* on page 483
29	—	Reserved			
30	Sn	MFC DMA list command stall-and-notify	Channel count for `MFC_RdListStallStat` changes from '0' to '1'.	A stall-and-notify flag set on a DMA list element has been encountered, the MFC has stopped executing the list, and the MFC has set the bit corresponding to the list's tag group in the `MFC_RdListStallStat` channel.	*Section 18.6.12* on page 484
31	Tg	MFC Tag-Group Status Update	Channel count for `MFC_RdTagStat` changes from '0' to '1'.	The tag-group status is currently available by reading the `MFC_RdTagStat` channel.	*Section 18.6.13* on page 486

18.2.2 Pending Event Register (Internal, SPE-Hidden)

The Pending Event register records all event occurrences, regardless of whether they are enabled. When an event occurs, the corresponding bit in the Pending Event register is set to 1. The Pending Event register has the layout shown in *Table 18-1* on page 464, but it is not directly accessible to SPE software. The contents of the register are, however, visible to privileged PPE software, using the SPU Channel Access Facility described in *Section 12.4.4* on page 356.

Since the Pending Event register reflects all pending events, regardless of the value of the SPU Write Event Mask, events that are pending but not enabled by the SPU Write Event Mask are nevertheless latched in Pending External Event. Consequently, if the SPU Write Event Mask is changed to enable a pending event, it will be reported in the SPU Read Event Status channel

immediately, regardless of how long the event has been pending. For this reason, the initialization code for event handling in SPE software should first clear any pending events (see *Section 18.2.5* on page 467) to prevent the raising of stale or phantom events.

18.2.3 SPU Read Event Status

Reading the SPU Read Event Status channel (`SPU_RdEventStat`) reports events that are both enabled by the SPU Write Event Mask and pending (that is, latched in Pending Event register) at the time of the channel read. Any event bit whose corresponding enable is cleared to ‘0’ in the SPU Write Event Mask will read as ‘0’ in the SPU Read Event Status. In other words, when software reads SPU Read Event Status, the value of the SPU Write Event Mask is logically ANDed with the value in the Pending Event register.

18.2.3.1 *Read Channel and Read Channel Count*

A read-channel-count (**rchcnt**) instruction to the SPU Read Event Status channel can return either ‘0’ or ‘1’, depending on the channel’s state, whereas reading the channel count of any of the other three event-management channels always returns ‘1’.

A **rchcnt** to SPU Read Event Status returns ‘0’ if no enabled events are pending, and it returns ‘1’ if enabled events have been raised since the last read of the status. A channel count of ‘1’ indicates that at least one event is pending (corresponding bit set in SPU Pending Event) and enabled (corresponding bit set in SPU Write Event Mask). A channel count of ‘0’ indicates no events are both enabled and pending.

Thus, SPE software can poll for pending, enabled events by reading the channel count, or it can execute a **rdch** instruction. A **rdch** when the channel count is ‘0’ causes the SPE to stall until at least one enabled event is pending. This stall behavior allows software to implement a *wait on external event* function. A further benefit is that the SPU uses significantly less power when stalled because most logic gates in the SPU stop switching. To avoid stalling, software can read the channel count before deciding whether to read the channel.

The channel count for SPU Read Event Status is set to ‘1’ for the following conditions:

- An event occurs when the corresponding bit in the SPU Write Event Mask has a value of ‘1’ (the event is enabled when the event occurs).
- At least one event is pending and enabled following a write of the SPU Write Event Mask (an event is already pending when the corresponding bit is set to ‘1’ by a write to the Mask). The previous state of the SPU Write Event Mask has no effect.
- At least one event is both enabled and pending after a write of the SPU Write Acknowledgement channel (the value written to SPU Write Acknowledgement did not acknowledge all events that were pending and enabled at the time of the write).

18.2.3.2 *Branch Condition*

The channel count of the SPU Read Event Status channel is used as the condition in the Branch Indirect and Set Link if External Data (**`bisled`**) instruction. If the channel count for SPU Read Event Status is ‘0’, the branch is not taken; if the channel count is ‘1’, the branch is taken. SPE software can execute a **`bisled`** instruction periodically to poll for events, and if **`bisled`** instructions

are executed frequently enough, software can approximate asynchronous event detection. See *Section 18.5.1* on page 470. For more information on the **`bisled`** instruction, see *Table B-1* on page 762 and the *Synergistic Processor Unit Instruction Set Architecture* document.

18.2.3.3 *SPU Interrupts*

If SPU interrupts are enabled (the IE bit in `SPU_RdMachStat` is set to ‘1’), an interrupt is taken when the channel count for SPU Read Event Status changes from ‘0’ to ‘1’, which means that one or more enabled events are pending. The only way to enable or disable interrupts while the SPU is running is to execute an indirect branch with either the `D` or `E` feature bit set. The SPU interrupt can be enabled or disabled while the SPU is stopped by setting the I bit in the SPU Next Program Counter (`SPU_NPC`) register. See the *Synergistic Processor Unit Instruction Set Architecture* for details.

18.2.4 SPU Write Event Mask

An SPE program determines which events it will monitor by setting the appropriate bits in the SPU Write Event Mask channel (`SPU_WrEventMask`). The value written to the SPU Write Event Mask determines which pending events will affect the value returned by a read of the SPU Read Event Status channel. When a bit in the Mask has a value of ‘1’, the corresponding event is said to be enabled. The value written to this channel is retained until a subsequent write changes the value. The current value of the SPU Write Event Mask can be read from the SPU Read Event Mask channel.

18.2.5 SPU Write Event Acknowledgement

Before SPE software services the events reported in SPU Read Event Status, software should write a value to the SPU Write Event Acknowledge channel (`SPU_WrEventAck`) to acknowledge (clear) the events that will be processed. The bits set to ‘1’ in the value that is written to SPU Write Event Acknowledge cause the corresponding bits in the hidden Pending Event register to be cleared to ‘0’. Thus, after software writes acknowledgment bits, events of the same type can be reported again. Clearing all pending and enabled events has no effect on the channel count for SPU Read Event Status.

All events are recorded in the hidden Pending Event register, regardless of the SPU Write Event Mask value. A pending, enabled event will continue to be reported in SPU Read Event Status until it is acknowledged—that is, cleared by writing an appropriate value to the SPU Write Event Acknowledgement channel; each bit position that corresponds to a pending event must be set to ‘1’.

The bit for a pending event can be cleared in the hidden SPU Pending Event register with a write to the SPU Event Acknowledgement channel even if the event is disabled. A disabled event that becomes pending and is subsequently acknowledged (cleared) is not reflected in the value read from the SPU Read Event Status channel. An event that has been acknowledged is resampled to allow future occurrences of the event to be recorded in the hidden SPU Pending Event register.

Enabling a currently pending but disabled event by writing an appropriate mask value to the SPU Write Event Mask channel results in an update of the SPU Read Event Status channel count (and an SPU interrupt, if enabled).

Acknowledging an event before it occurs results in a software-induced deadlock. Software should be careful in clearing unreported events.

18.2.6 SPU Read Event Mask

Software uses the SPU Read Event Mask channel (`SPU_RdEventMask`) to read the state of the SPU Write Event Status Mask. The SPU Read Event Mask always returns the data last written to the mask. Instead of keeping software shadow copies of the Event Status Mask, SPE software can read the state of the mask directly. SPE context save and restore operations can also use this channel.

18.3 SPU Interrupt Facility

The SPE can monitor event status through the SPU Read Event Status (`SPU_RdEventStat`) channel. Software can process the event status in the `SPU_RdEventStat` channel by polling, using the **bisled** instruction or by enabling the SPU interrupt facility. When enabled, the interrupt facility causes the SPE to execute the interrupt handler located at LS address x‘0’ when the `SPU_RdEventStat` channel has nonzero count.

The SPU interrupt can be enabled in one of two ways:

- Setting the E bit in the indirect-branch instruction (see the *Synergistic Processor Unit Instruction Set Architecture* manual).
- Setting bit 31 of the SPU Next Program Counter Register (`SPU_NPC`) prior to running the SPE.

An SPU interrupt is taken if the interrupt is enabled and the count in the `SPU_RdEventStat` channel has transitioned from ‘0’ to ‘1’. This transition is edge-triggered, based on the ORing of events in the `SPU_RdEventStat` channel, masked by bits in the SPU Write Event Mask (`SPU_WrEventMask`) channel, and the acknowledgment bits in the SPU Write Event Acknowledgment (`SPU_WrEventAck`) channel. For a logical representation of the SPE event support and the descriptions of the `SPU_RdEventStat` channel, the `SPU_WrEventMask` channel, and the `SPU_WrEventAck` channel, see the *Cell Broadband Engine Architecture* document. See the *Synergistic Processor Unit Instruction Set Architecture* document for information about using indirect branches for entering and returning from the interrupt handler.

An interrupt sequence might go through the following steps:

1. Enable interrupts as described above.
2. The program runs until conditions cause the channel count for the `SPU_RdEventStat` channel to transition from '0' to '1'; then, the interrupt is taken.
3. Interrupts are acknowledged through internal logic as follows:
   a. Internal logic disables interrupts.
   b. Any blocked channel access is preempted.
   c. The program counter goes to LS address 0.
   d. The return address is stored in SRR0.
4. Read the `SPU_RdEventStat` channel to see the status of events (the channel count for the `SPU_RdEventStat` channel bit transitions to '0').

5. Acknowledge all events to be processed (write to the `SPU_WrEventAck` channel).
6. Process events (for example, mailbox events).
7. Execute an interrupt return (**iret**) instruction, which branches to the return address held in SRR0. The SPU interrupt can be re-enabled using the E bit of the **iret** instruction.

Nested interrupts are supported through reading the SPU Read State Save-and-Restore (`SPU_RdSRR0`) channel and writing the SPU Write State Save-and-Restore (`SPU_WrSRR0`) channel.

The value written to the `SPU_WrSRR0` channel is not immediately available to the **iret** instruction. This write must be synchronized by execution of a channel **sync** instruction (with the C bit set in the instruction) before executing an **iret** instruction.

For details about handling SPU interrupts, see *Section 18.5.2 Asynchronous Event Handling Using Interrupts* on page 471.

18.4 Interrupt Address Save-and-Restore Channels

Two channels are used for saving and restoring the interrupt return address: SPU Read State Save-and-Restore (`SPU_RdSRR0`) and SPU Write State Save-and-Restore (`SPU_WrSRR0`).

18.4.1 SPU Read State Save-and-Restore

When an SPE takes an interrupt, the current program counter is stored in the SPU Read State Save-and-Restore (`SPU_RdSRR0`) channel and execution continues are address 0. The saved program counter can be acquired by reading `SPU_RdSRR0`.

SPU interrupts are precise; that is, all instructions in the program flow prior to interrupt will have completed execution, and no subsequent instructions will have begun.

18.4.2 SPU Write State Save-and-Restore

SPE software can write to the SPU Write State Save-and-Restore (`SPU_WrSRR0`) channel to set the return address for an interrupt-return (**iret**) instruction. This channel should only be written when SPU interrupts are disabled, and after it is written, the **syncc** instruction should be executed before the interrupt-return instruction. The **syncc** instruction (**sync** with the channel-feature bit set) makes sure channel writes complete before any subsequent instructions can execute.

18.4.3 Nested Interrupts Using SPU Write State Save-and-Restore

The SPE architecture does not have the ability to assign priorities to event interrupts, but nested interrupts can help software approximate prioritized interrupts by allowing a low-priority event interrupt handler to be interrupted by a higher-priority event.

To implement nested interrupts, SPU interrupt handling code can save the SPU Read State Save-and-Restore value before re-enabling interrupts. After handling the initial interrupt, SPU interrupt code must:

1. Disable interrupts.

2. Write to SPU Write State Save-and-Restore to restore the initial interrupt return address.
3. Execute a **syncc** instruction (**sync** with the channel feature-bit set) to force the channel write to complete.
4. Execute an **irete** instruction to return from the initial interrupt and re-enable interrupts.

18.5 Event-Handling Protocols

SPE software can be written to handle events either synchronously or asynchronously. Synchronous event handling can be implemented by polling or stalling; asynchronous event handling is implemented with interrupts.

18.5.1 Synchronous Event Handling Using Polling or Stalling

SPE software can use polling to recognize an enabled and pending event either by reading the channel count of SPU Read Event Status, channel 0, or by executing the **bisled** instruction.

When no enabled events are pending, the channel count of SPU Read Event Status is ‘0’; when one or more enabled events are pending, the count is ‘1’. Thus, SPE event-polling code can implement a loop such as the one shown in the code fragment below:

```
do {
    /* Perform work between checks of the
     * channel count, as appropriate.
     */
} while (!spu_readchcnt(SPU_RdEventStat));
```

A programmer can use the **bisled** instruction or the **spu_bisled** intrinsic to test the SPU Read Event Status channel count. The instruction or intrinsic jumps to the indirect branch target when the channel count is ‘1’ but falls through when the count is ‘0’.

As shown in *Table 17-2* on page 443, the SPU Read Event Status channel is a blocking channel. If SPE software executes a **rdch** instruction on the channel, the **rdch** instruction will stall when there are no enabled events pending. This provides a simple way to implement a general “wait on external event” capability as illustrated in the following simple code fragment:

```
unsigned int event;

/* Block until event is available. */
event = spu_readch(SPU_RdEventStat);
```

SPE software that is concerned with only a single type of event, such as a mailbox or signal arrival, can read the associated channel or channel count to achieve stalling or polling behavior if the associated channel can block. Thus, if SPE software is concerned only with events on the SPU Signal Notification 1 channel, it can poll the channel count for `SPU_RdSigNotify1` with the **rchcnt** instruction, or software can stall by reading `SPU_RdSigNotify1`, a blocking channel, with a **rdch** instruction. If SPE software is concerned only with incoming mailbox events, it can stall by

blocking on a **rdch** from `SPU_RdInMbox`. Polling or stalling on a specific channel instead of on the SPU Read Event Status channel can save a small amount of overhead (setting up the event mask and writing the event acknowledgment).

18.5.2 Asynchronous Event Handling Using Interrupts

When SPU interrupts are enabled and at least one event is pending in the SPU Read Event Status channel, SPE hardware takes the following steps to recognize the event interrupt:

- Interrupts are disabled.
- The address of the next instruction that would have executed is saved in Save-Restore Register 0.
- SPE hardware begins executing instructions from address 0 in the SPE's LS.

The only way SPE software can enable and disable interrupts under program control is by using the `D` and `E` feature bits in indirect-branch instructions; SPE software cannot write directly to SPE machine status. *Table 18-3* shows the instructions that can be qualified with the `D` and `E` feature bits to enable or disable interrupts. See the *Synergistic Processor Unit Instruction Set Architecture* for details.

Table 18-3. Indirect-Branch Instructions

Mnemonic	Description
bi	Branch Indirect
bisl	Branch Indirect and Set Link
bisled	Branch Indirect and Set Link If External Data
biz	Branch Indirect If Zero
binz	Branch Indirect If Not Zero
bihz	Branch Indirect If Zero Halfword
bihnz	Branch Indirect If Not Zero Halfword
iret	Interrupt Return

These instructions leave the interrupt-enable status unchanged unless either the `D` or `E` feature bit is specified (an instruction with both feature bits causes undefined behavior).

When one of these branches is taken with an interrupt feature bit set, the interrupt-enable status change takes effect before the target instruction is executed.

SPE hardware supports one hardwired interrupt vector for all interrupts; this vector is to address 0 in LS. This single interrupt handler at address 0 must determine which events have occurred and process them appropriately.

If an SPE application needs nested interrupt handling, the handler at address 0 must save the interrupt return address from Read Save-Restore Register 0, prepare for a possible nested interrupt (for example, save other volatile state), and enter an interruptible event handler with interrupts enabled (that is, execute an indirect branch with the `E` feature bit to the interruptible event handler).

When (possibly nested) event processing is complete, an interruptible interrupt handler can return to the instruction stream it interrupted in one of two ways:

- By writing the return address to SPU Write State Save-and-Restore and executing a **syncc** or **iret** instruction pair (**syncc** is needed to force the channel write before **iret** executes).
- By loading the return address into an SPE register and executing a **bie** instruction (or other suitable indirect branch).

In either case, the **iret** or **bi** instruction should have the `E` feature bit set to re-enable event interrupts.

18.5.3 Protecting Critical Sections from Interruption

When asynchronous interrupts are enabled, an interrupt can happen at any time while application code is running. Since applications may contain critical sections of code that must not be interrupted, programs must disable interrupts while a critical section is executing.

Critical sections arise in typical SPE application programs. For example, an application that enqueues DMA commands must do so with a sequence of instructions that forms the complete DMA command. When both application and interrupt handler code enqueue DMA commands, the sequence of channel writes that forms the DMA command must not be interrupted; the channel writes for a single DMA command must all complete before the first channel write for the next DMA command happens.

In high-level-language code, the SPE intrinsics `spu_idisable()` and `spu_ienable()` can be used to surround a critical section. In assembler code, indirect branches must be used to disable and enable interrupts. The code fragments below show one way to implement disable and enable operations in assembler; in fact, these are the code sequences in the intrinsics `spu_idisable()` and `spu_ienable()`.

```
idisable:
    ila    $Rt, next_inst  # form address of instruction after bie
    bid    $Rt             # branch to next_inst and disable interrupts
next_inst:

[code inside critical section]

ienable:
    ila    $Rt, next_inst  # form address of instruction after bie
    bie    $Rt             # branch to next_inst and enable interrupts
next_inst:
```

Note: *A branch to the next sequential instruction will not be mispredicted and need not be hinted.*

18.6 Event-Specific Handling Guidelines

Most of the semantics of event handling will be application specific and therefore beyond the scope of this discussion, but all applications should follow general guidelines for event-handling protocol and satisfy the minimum requirements for each possible event. Examples of event handlers with application-specific code are given later in this chapter (see *Section 18.10.1* on page 497).

For each event, the conditions that lead to it being raised are discussed in the following sections. One of the necessary conditions for any event to be raised—that event be enabled in the SPU Write Event Mask—is assumed but not mentioned in the discussions.

18.6.1 Protocol With Multiple Events Enabled

When multiple events are enabled (when the SPU Write Event Mask has more than one bit set to '1'), SPE software should follow a simple procedure to prevent the occurrence of spurious (phantom) events. Before SPE software performs any event-specific actions, it must disable the events that will be processed and then acknowledge them. This allows the MFC hardware to register new occurrences of the events as soon as possible. After all event-specific processing is completed, SPE software restores the event mask. This allows the new occurrences of the processed events to be recognized in the SPU Read Event Status channel.

The steps of the common protocol, when multiple events are enabled, are shown below:

1. Save SPU Read Event Status (`SPU_RdEventStat`) in `status`.
2. Save SPU Read Event Mask (`SPU_RdEventMask`) in `mask`.
3. Disable events: write (`mask & ~status`) to SPU Write Event Mask (`SPU_WrEventMask`).
4. Acknowledge events: write `status` to SPU Write Event Acknowledge (`SPU_WrEventAck`).
5. Process events (see *Table 18-4* on page 473 and individual event sections below).
6. Restore the event mask: write `mask` to SPU Write Event Mask (`SPU_WrEventMask`).

A common implementation of this protocol is to perform Steps 1 through 4 and 6 in a first-level event handler routine and to encapsulate the code for handling each specific event in second-level event handling routines. With this organization, the first-level event handler sets up the environment and then calls each second-level event handler routines that corresponds to a bit set to '1' in the event status. This two-level organization is used in the interrupt-driven event handling examples discussed in *Section 18.7* on page 486, *Section 18.8* on page 493, *Section 18.9* on page 495, and *Section 18.10.1* on page 497.

The following sections describe the recommended protocols for handling specific events. *Table 18-4* on page 473 lists the events along with protocol summaries.

Table 18-4. SPE Event Handling Protocol Summaries (Page 1 of 2)

Field Name	Bit	Event Name	Handling Protocol	Reference
Ms	19	Multisource Synchronization	• Release resource waiting on synchronization.	*Section 18.6.2* on page 474
A	20	Privileged Attention	• Perform high-priority response such as stop-and-signal, mailbox communication, or update in-memory status.	*Section 18.6.3* on page 476
Lr	21	Lock-Line Reservation Lost	• If lock still being used, re-issue lock and act on new data.	*Section 18.6.4* on page 476
S1	22	SPU Signal-notification 1 Available	• Read channel count of `SPU_RdSigNotify1`; if '0', handler done. • Read `SPU_RdSigNotify1` and process signal data.	*Section 18.6.5* on page 477

Table 18-4. SPE Event Handling Protocol Summaries (Page 2 of 2)

Field Name	Bit	Event Name	Handling Protocol	Reference
S2	23	SPU Signal-notification 2 Available	• Read channel count of `SPU_RdSigNotify2`; if '0', handler done. • Read `SPU_RdSigNotify2` and process signal data.	*Section 18.6.6* on page 478
Le	24	SPU Outbound Mailbox Available	• Read channel count of `SPU_WrOutMbox`; if '0', handler done. • Write `SPU_WrOutMbox` with mail box data.	*Section 18.6.7* on page 479
Me	25	SPU Outbound Interrupt Mailbox Available	• Read channel count of `SPU_WrOutIntrMbox`; if '0', handler done. • Write `SPU_WrOutIntrMbox` with mail box data.	*Section 18.6.8* on page 480
Tm	26	SPU Decrementer	• Read `SPU_RdDec`; if negative, this is time delay from event to response (additional time base ticks since decrementer count expired). • Write `SPU_WrDec` with new timer interval count (optionally reduced by the time delay discovered above).	*Section 18.6.9* on page 480
Mb	27	SPU Read Inbound Mailbox Available	• Read channel count of `SPU_RdInMbox`; if '0', handler done. • Read `SPU_RdInMbox` and process mail box data. • Write `SPU_WrEventAck` with `Mb` bit set (acknowledge each mail box entry). • Repeat.	*Section 18.6.10* on page 482
Qv	28	MFC SPU Command Queue Available	• Read channel count of `MFC_Cmd`; if '0', handler done. • Enqueue DMA command w/ sequence of channel writes. • If no more commands to enqueue, exit. • Write `SPU_WrEventAck` with `Qv` bit set. • Repeat.	*Section 18.6.11* on page 483
Sn	30	MFC DMA List Command Stall-and-Notify	• Read `MFC_RdListStallStat` to get stalled tag groups. • Choose a stalled tag group. • Process the stalled tag group. • Write `MFC_WrListStallAck` for the stalled tag group. • Repeat previous three steps until all stalled tag groups processed.	*Section 18.6.12* on page 484
Tg	31	MFC Tag-Group Status Update	• Read channel count of `MFC_RdTagStat`; if '0', handler done. • Read `MFC_RdTagStat` to get new tag-group status.	*Section 18.6.13* on page 486

18.6.2 Procedure for Handling the Multisource Synchronization Event

A Multisource Synchronization Event is the expected response after SPE software writes to MFC Write Multisource Synchronization Request, channel 9 (`MFC_WrMSSyncReq`). SPE software can write any value to start the request, but writing a value of '0' ensures compatibility with future

architectural revisions. The occurrence of a multisource synchronization event indicates that all the write transfers directed at the SPE's MFC that were started prior to the multisource synchronization request (the write to `MFC_WrMSSyncReq`) have completed.

SPE software must use the multisource synchronization facility when cumulative ordering is needed across address domains. Cumulative ordering is the ordering of storage accesses performed by multiple sources (that is, two or more processor elements) with respect to another processor element.

See *Section 20.1.5 MFC Multisource Synchronization Facility* on page 567 for a description of multisource synchronization. Graphic flowcharts of the possible software sequences are given in *Figure 20-2* on page 569, *Figure 20-3* on page 571, and *Figure 20-4* on page 572.

SPE software uses multisource synchronization when it needs data that depends on the completion of a chain of storage operations that other processor elements perform. Thus, when the multisource synchronization event occurs, the action taken by SPE software is application-dependent. Two typical meanings of the event are that (1) an input buffer is completely filled and ready for SPE processing, and (2) that an output buffer is no longer in use and can be reallocated for future use.

To ensure the multisource synchronization event will be raised in response to a specific request, SPE software should perform an initial acknowledgement of the event (write '1' to the `Ms` bit of the `SPU_WrEventAck` channel) before enabling it (write '1' the `Ms` bit of the `SPU_WrEventMask` channel). This initial clearing of a possible stale (phantom) pending multisource synchronization event need only happen once during SPE application start-up, because adherence to the protocols outlined here will ensure that each event occurrence is properly matched with its acknowledgement.

The procedure for handling the Multisource Synchronization Event is as follows:

1. Perform a read channel (**rdch**) instruction to the SPU Read Event Mask Channel and save the data in a "mask".[36]
2. Mask the event by issuing a write channel (**wrch**) instruction to the SPU Write Event Mask Channel with `SPU_WrEventMask[Ms]` set to '0'.[36]
3. Acknowledge the event by performing a write channel (**wrch**) instruction to the SPU Write Event Acknowledgment Channel with `SPU_WrEventAck[Ms]` set to '1'.[36]
4. Perform the application-specific function in response to the completion of a pending multisource synchronization operation.

   This would typically indicate that the data in a particular buffer has been completely updated, or that a buffer area is no longer in use.
5. Exit the Multisource Synchronization Event handler.
6. Restore the "mask" by issuing a write channel (**wrch**) instruction to the SPU Write Event Mask Channel with `SPU_WrEventMask[Ms]` set to '1'.[36]
7. Exit the general event handler.[36]

36. When multiple events are enabled, a common handler should be used to save the current event mask, mask all events that are to be handled in one channel write, and acknowledge all events that are to be handled in a single channel write. Then each event specific handler should be invoked to handle the event. The common handler should then restore the current event mask from the saved value and exit. This technique minimizes the generation of spurious events.

18.6.3 Procedure for Handling the Privileged Attention Event

The Privileged Attention Event can be raised when a ‘1’ is written by the PPE to the SPU Privileged Control Register’s Attention Event Required bit. If the event is enabled when this bit is set to ‘1’, the event is raised.

The intended use of the Privileged Attention Event is to inform SPE software that privileged software running on the PPE requires high-priority action. Examples of such actions are an SPE context switch and reporting SPE status to PPE software. Thus, the handler for this event will contain or call the functions that implement the high-priority action.

Since the Privileged Attention Event communicates only one bit of information—the event itself—an application-specific protocol must be implemented if additional information must be passed between the PPE and the SPE. For example, information about the type of attention requested can be passed in a memory location or as a value written to a signal or mailbox channel.

The procedure for handling the Privileged Attention Event is:

1. Perform a read channel (**rdch**) instruction to the SPU Read Event Mask Channel and save the data in “mask”.[37]
2. Mask the event by issuing a write channel (**wrch**) instruction to the SPU Write Event Mask Channel with `SPU_WrEventMask[A]` set to ‘0’.[37]
3. Acknowledge the event by performing a write channel (**wrch**) instruction to the SPU Write Event Acknowledgment Channel with `SPU_WrEventAck[A]` set to ‘1’.
4. Perform the application-specific function in response to a Privileged Attention Event.

   This can be used to signal that a yield of the SPU is being requested or some other action. An application or operating system-specific response to the Privileged Attention Event should be issued, such as stop and signal, SPU Inbound mailbox write, SPU Outbound Interrupt mailbox write, or an update of a status in system or I/O memory space.
5. Exit the Privileged Attention Event handler.
6. Restore the “mask” by issuing a write channel (**wrch**) instruction to the SPU Write Event Mask Channel with `SPU_WrEventMask[A]` set to ‘1’.[37]
7. Exit the general event handler.[37]

18.6.4 Procedure for Handling the Lock-Line Reservation Lost Event

This event is raised when a get lock-line and reserve (**getllar**) command is issued, and the reservation is reset due to the modification of data in the same lock line by an outside entity. A lock-line reservation is reset because the snoop hardware in the SPE’s atomic unit (atomic-update cache) detects a data modification by the outside entity.This event will not be raised due to a reservation reset by any local SPE action.

---

37. When multiple events are enabled, a common handler should be used to save the current event mask, mask all events that are to be handled in one channel write, and acknowledge all events that are to be handled in a single channel write. Then each event specific handler should be invoked to handle the event. The common handler should then restore the current event mask from the saved value and exit. This technique minimizes the generation of spurious events.

When SPE software executes a **getllar** command, the atomic unit caches the 128-byte line and marks the status of the line as reserved. Usually, the reservation will be reset when SPE software executes a matching **putllc** or **putlluc** to complete an atomic operation (see *Section 20 Shared-Storage Synchronization* on page 551 for a complete discussion of atomic operations using atomic lock-line commands). When the reservation is reset by a matching **put** operation, the Lock-Line Reservation Lost event will not be raised. If another processor element or device modifies the data in the reserved lock-line, the reservation will be lost and the event will be raised if it is enabled.

The handler for this event should decide if the lock-line reservation should be renewed, and if so, issue a duplicate **getllar** command. There is no need to read the atomic command status to confirm that the reservation was lost; the fact that the event was received means that the reservation was lost.

The procedure for handling the Lock-Line Reservation Lost Event is:

1. Perform a read channel (**rdch**) instruction to the SPU Read Event Mask Channel and save the data in “mask”.[38]
2. Mask the event by issuing a write channel instruction to the SPU Write Event Mask Channel with `SPU_WrEventMask[Lr]` set to ‘0’.[38]
3. Acknowledge the event by performing a write channel (**wrch**) instruction to the SPU Write Event Acknowledgment Channel with `SPU_WrEventAck[Lr]` set to ‘1’.[38]
4. Perform the application-specific function in response to system modification of data in the lock line area.

   This is usually started by checking a software structure in memory to determine if a lock line is still being monitored. If it is still being “waited on,” then the next step would typically consist of issuing a **getllar** command to the same lock line area that was modified to obtain the new data and then act on that data.
5. Exit the Lock Line Reservation Lost Event handler.
6. Restore the “mask” by issuing a write channel (**wrch**) instruction to the SPU Write Event Mask Channel with `SPU_WrEventMask[Lr]` set to ‘1’.[38]
7. Exit the general event handler.[38]

18.6.5 Procedure for Handling the Signal-Notification 1 Available Event

The Signal-Notification 1 available event is raised when another processor or device has written to an empty SPU Signal Notification 1 register. The event occurrence also means that the channel count has changed from ‘0’ to ‘1’; thus, the SPU can read the value of the signal without stalling.

38. When multiple events are enabled, a common handler should be used to save the current event mask, mask all events that are to be handled in one channel write, and acknowledge all events that are to be handled in a single channel write. Then each event specific handler should be invoked to handle the event. The common handler should then restore the current event mask from the saved value and exit. This technique minimizes the generation of spurious events.

To account for the possibility of a phantom event, the procedure for handling a Signal-Notification 1 Available event should start by reading the channel count for the SPU Signal-Notification 1 channel (`SPU_RdSigNotify1`). If the count is ‘0’, the event was a phantom event and no signal data is waiting to be read. If the count is not ‘0’, the handler should read `SPU_RdSigNotify1` to get the signal data and reset the signal data to ‘0’ in preparation for the next signal.

The procedure for handling the SPU Signal-Notification 1 Available Event is:

1. Perform a read channel (**rdch**) instruction to the SPU Read Event Mask Channel and save the data in “mask”.[39]
2. Mask the event by issuing a write channel (**wrch**) instruction to the SPU Write Event Mask Channel with `SPU_WrEventMask[S1]` set to ‘0’.[39]
3. Acknowledge the event by performing a write channel (**wrch**) instruction to the SPU Write Event Acknowledgment Channel with `SPU_WrEventAck[S1]` set to ‘1’.[39]
4. Obtain the channel count by issuing a read channel count (**rchcnt**) instruction to the SPU Signal Notification 1 Channel.
5. If the channel count is ‘0’, skip to step 7.
6. Read the signal data by issuing a read channel (**rdch**) instruction to the SPU Signal Notification 1 Channel (`SPU_RdSigNotify1`).
7. Exit the Signal Notification 1 handler.
8. Restore the “mask” by issuing a write channel (**wrch**) instruction to the SPU Write Event Mask Channel with `SPU_WrEventMask[S1]` set to ‘1’.[39]
9. Exit the general event handler.[39]

18.6.6 Procedure for Handling the Signal-Notification 2 Available Event

The Signal-Notification 2 available event is raised when another processor or device has written to an empty SPU Signal Notification 2 register. The event occurrence also means that the channel count has changed from ‘0’ to ‘1’; thus, the SPU can read the value of the signal without stalling.

To account for the possibility of a phantom event, the procedure for handling a Signal-Notification 2 Available event should start by reading the channel count for the SPU Signal-Notification 2 channel (`SPU_RdSigNotify2`). If the count is ‘0’, the event was a phantom event and no signal data is waiting to be read. If the count is not ‘0’, the handler should read `SPU_RdSigNotify2`, which will get the signal data and reset the signal data to ‘0’ in preparation for the next signal.

The procedure for handling the SPU Signal-Notification 2 Available Event is:

1. Perform a read channel (**rdch**) instruction to the SPU Read Event Mask Channel and save the data in “mask”.[39]
2. Mask the event by issuing a write channel (**wrch**) instruction to the SPU Write Event Mask Channel with `SPU_WrEventMask[S2]` set to ‘0’.[39]

39. When multiple events are enabled, a common handler should be used to save the current event mask, mask all events that are to be handled in one channel write, and acknowledge all events that are to be handled in a single channel write. Then each event specific handler should be invoked to handle the event. The common handler should then restore the current event mask from the saved value and exit. This technique minimizes the generation of spurious events.

3. Acknowledge the event by performing a write channel (**wrch**) instruction to the SPU Write Event Acknowledgment Channel with `SPU_WrEventAck[S2]` set to a '1'.[39]
4. Obtain the channel count by issuing a read channel count (**rchcnt**) instruction to the SPU Signal Notification 2 Channel.
5. If the channel count is '0', skip to step 7.
6. Read the signal data by issuing a read (**rdch**) channel instruction to the SPU Signal Notification 2 Channel.
7. Exit the Signal Notification 2 handler.
8. Restore the "mask" by issuing a write channel (**wrch**) instruction to the SPU Write Event Mask Channel with `SPU_WrEventMask[S2]` set to '1'.[39]
9. Exit the general event handler.[39]

18.6.7 Procedure for Handling the SPU Write Outbound Mailbox Available Event

The SPU Write Outbound Mailbox Available event is raised when a PPE or other device reads from the SPU Write Outbound Mailbox and the mailbox has available data. When the mailbox data is read by the PPE or device, the channel count for the SPU Write Outbound Mailbox channel (`SPU_WrOutMbox`) changes from '0' to '1', which means the SPU can write `SPU_WrOutMbox` without stalling.

To account for the possibility of a phantom event, the procedure for handling a SPU Write Outbound Mailbox Available event should start by reading the channel count for `SPU_WrOutMbox`. If the count is '0', the event was a phantom event and the mailbox is still full. If the count is not '0', the handler should write `SPU_WrOutMbox` with the next mailbox data to send to the PPE.

The procedure for handling the SPU Write Outbound Mailbox Available Event is:

1. Perform a read channel (**rdch**) instruction to the SPU Read Event Mask Channel and save the data in "mask".[40]
2. Mask the event by issuing a write channel (**wrch**) instruction to the SPU Write Event Mask Channel with `SPU_WrEventMask[Le]` set to '0'.[40]
3. Acknowledge the event by performing a write channel (**wrch**) instruction to the SPU Write Event Acknowledgment Channel with `SPU_WrEventAck[Le]` set to '1'.[40]
4. Obtain the channel count by issuing a read channel count (**rchcnt**) instruction to the SPU Write Outbound Mailbox Channel.
5. If the channel count is '0', skip to step 7.
6. Write a new mailbox data entry by issuing a write channel (**wrch**) instruction to the SPU Write Outbound Mailbox Channel.
7. Exit the SPU Outbound Mailbox handler.
8. Restore the "mask" by issuing a write channel (**wrch**) instruction to the SPU Write Event Mask Channel with `SPU_WrEventMask[Le]` set to '1'.[40]

---

40. When multiple events are enabled, a common handler should be used to save the current event mask, mask all events that are to be handled in one channel write, and acknowledge all events that are to be handled in a single channel write. Then each event specific handler should be invoked to handle the event. The common handler should then restore the current event mask from the saved value and exit. This technique minimizes the generation of spurious events.

9. Exit the general event handler.[40]

18.6.8 Procedure for Handling the SPU Write Outbound Interrupt Mailbox Available Event

The SPU Write Outbound Interrupt Mailbox Available event is raised when a PPE or other device reads from the SPU's PPE interrupt mailbox and the mailbox has available data. When the mailbox data is read by the PPE or device, the channel count for the SPU Write Outbound Interrupt Mailbox channel (`SPU_WrOutIntrMbox`) changes from '0' to '1', which means the SPU can write `SPU_WrOutIntrMbox` without stalling.

To account for the possibility of a phantom event, the procedure for handling a SPU Write Outbound Interrupt Mailbox Available event should start by reading the channel count for `SPU_WrOutIntrMbox`. If the count is '0', the event was a phantom event and the mailbox is still full. If the count is not '0', the handler should write `SPU_WrOutIntrMbox` with the next mailbox data to send to the PPE.

The procedure for handling the SPU Write Outbound Interrupt Mailbox Available Event is:

1. Perform a read channel (**rdch**) instruction to the SPU Read Event Mask Channel and save the data in the "mask".[41]
2. Mask the event by issuing a write channel (**wrch**) instruction to the SPU Write Event Mask Channel with `SPU_WrEventMask[Me]` set to '0'.[41]
3. Acknowledge the event by performing a write channel (**wrch**) instruction to the SPU Write Event Acknowledgment Channel with `SPU_WrEventAck[Me]` set to a '1'.[41]
4. Obtain the channel count by issuing a read channel count (**rchcnt**) instruction to the SPU Write Outbound Interrupt Mailbox Channel.
5. If channel count is '0', skip to step 7.
6. Write a new mailbox data entry by issuing a write channel (**wrch**) instruction to the SPU Write Outbound Interrupt Mailbox Channel.
7. Exit the SPU Outbound Interrupt Mailbox Available handler.
8. Restore the "mask" by issuing a write channel (**wrch**) instruction to the SPU Write Event Mask Channel with `SPU_WrEventMask[Me]` set to '1'.[41]
9. Exit the general event handler.[41]

18.6.9 Procedure for Handling the SPU Decrementer Event

The SPU Decrementer event is raised when the decrementer value changes from '0' to negative; that is, the SPU Decrementer event is raised when the most-significant bit of the decrementer changes from '0' to '1'.

41. When multiple events are enabled, a common handler should be used to save the current event mask, mask all events that are to be handled in one channel write, and acknowledge all events that are to be handled in a single channel write. Then each event specific handler should be invoked to handle the event. The common handler should then restore the current event mask from the saved value and exit. This technique minimizes the generation of spurious events.

The decrementer maintains a signed, twos-complement value in a 32-bit down counter. The decrementer counts down at the rate of the PPE time base; all decrementers in the SPUs and the PPE count down at the same rate. The value of the decrementer can be read by SPE software through SPU Read Decrementer (`SPU_RdDec`) and set through SPU Write Decrementer (`SPU_WrDec`).

The SPU decrementer can run if the `Dh` (decrementer halt) bit in the `MFC_CNTL` register is cleared to '0' by PPE software (see *Section 13 Time Base and Decrementers* on page 375). If the `Dh` bit is set, SPE software cannot start the decrementer.

After an SPU Decrementer event becomes pending, it is possible that a significant amount of time may elapse before the event handler runs. Depending on the needs of the application program, it may be desirable to account for the delay in setting the next time interval to be measured by the decrementer.

To account for the delay, the event handler can read `SPU_RdDec` to get the current value of the decrementer. If the decrementer has been running during the delay, the value will be the twos-complement of the number of time-base ticks that have elapsed between the time the event was raised and the read of `SPU_RdDec`. To account for the time lapse in the next decrementer interval, the handler can simply add the desired decrementer count to the (negative) value read from `SPU_RdDec` and write the result to `SPU_WrDec`. Software will thus set the decrementer to a new count that is adjusted by the number of time-base ticks since the event was raised.

If the protocol given above in *Section 18.6.1* on page 473 is followed, however, the decrementer will stop when the acknowledgement step is executed (acknowledging the decrementer event when it is disabled stops the decrementer). In this case, the twos-complement value read from `SPU_RdDec` as described above may not account for all the time lapse between the event being raised and the execution of the decrementer event-handler (other handlers may run before the decrementer handler while the decrementer is stopped).

Thus, it may be necessary to either structure interrupt handling to run the decrementer event-handler first or not acknowledge the decrementer event immediately. If the handler follows a protocol that does not cause the decrementer to stop (the event is disabled but not acknowledged immediately or the event is enabled when it is acknowledged), then the twos-complement value read from `SPU_RdDec` will accurately represent the time delay between the event being raised and the handler running.

To start a stopped decrementer, SPE software writes `SPU_WrDec`. Thus, when the decrementer event-handler writes a value to `SPU_WrDec` for another time interval, the decrementer begins counting down immediately if it was stopped.

The procedure for handling the SPU Decrementer Event is:

1. Perform a read channel (**rdch**) instruction to the SPU Read Event Mask Channel and save the data in the "mask".[42]
2. Mask the event by issuing a write channel (**wrch**) instruction to the SPU Write Event Mask Channel with `SPU_WrEventMask[Tm]` set to '0'.[42]

42. When multiple events are enabled, a common handler should be used to save the current event mask, mask all events that are to be handled in one channel write, and acknowledge all events that are to be handled in a single channel write. Then each event specific handler should be invoked to handle the event. The common handler should then restore the current event mask from the saved value and exit. This technique minimizes the generation of spurious events.

3. Acknowledge the event by issuing a write channel (**wrch**) instruction to the SPU Write Event Acknowledgment Channel (`SPU_WrEventAck[Tm]` set to '1').
4. Read the decrementer value by issuing a read channel (**rdch**) instruction to the SPU Read Decrementer Channel. If this value is negative, it can be used to determine how much additional time has elapsed from the desired interval.
5. If a new timer event is desired, write (**wrch**) a new decrementer value to the SPU Write Decrementer Channel.
6. Exit the SPU decrementer event handler.
7. Restore the "mask" by issuing a write channel (**wrch**) instruction to the SPU Write Event Mask Channel with `SPU_WrEventMask[Tm]` set to '1'.[42]
8. Exit the general event handler.[42]

18.6.10 Procedure for Handling the SPU Read Inbound Mailbox Available Event

The SPU Read Inbound Mailbox event is raised when the PPE or other device writes data to the SPU Read Inbound Mailbox and the mailbox is empty; the data is accessible by the SPE through the SPU Read Inbound Mailbox (`SPU_RdInMBox`). When the `SPU_RdInMBox` channel count changes from '0' to non-zero, the event is raised.

When the PPE or other device writes data to the inbound mailbox, the channel count for the `SPU_RdInMBox` is incremented. The queue depth of the `SPU_RdInMBox` is four, and the maximum channel count is four. It is possible for a PPE or other device to overrun the SPU mailbox which causes the most-recently written value to be lost.

When the SPU Mailbox Available event handler executes, the SPU mailbox queue may have from one to four valid entries. Since the event is only raised when the `SPU_RdInMBox` channel count changes from '0' to non-zero, the handler must read and process all valid entries to prevent any entries from being ignored for an arbitrarily long period of time

The procedure for handling the SPU Read Inbound Mailbox Available Event is:

1. Perform a read channel (**rdch**) instruction to the SPU Read Event Mask Channel and save the data in "mask".[43]
2. Mask the event by issuing a write channel (**wrch**) instruction to the SPU Write Event Mask Channel with `SPU_WrEventMask[Mb]` set to '0'.[43]
3. Acknowledge the event by performing a write channel (**wrch**) instruction to the SPU Write Event Acknowledgment Channel with `SPU_WrEventAck[Mb]` set to '1'.[43]
4. Obtain a channel count by issuing a read channel count (**rchcnt**) instruction to the SPU Read Inbound Mailbox Channel.
5. If the channel count is '0', skip to step 8.
6. Read next mailbox data entry by issuing a read channel (**rdch**) instruction to the SPU Read Inbound Mailbox Channel (`SPU_RdInMbox`).

---

43. When multiple events are enabled, a common handler should be used to save the current event mask, mask all events that are to be handled in one channel write, and acknowledge all events that are to be handled in a single channel write. Then each event specific handler should be invoked to handle the event. The common handler should then restore the current event mask from the saved value and exit. This technique minimizes the generation of spurious events.

7. Return to step 3.
8. Exit the SPU inbound mailbox handler.
9. Restore the “mask” by issuing a write channel (**wrch**) instruction to the SPU Write Event Mask Channel with `SPU_WrEventMask[Mb]` set to ‘1’.[43]
10. Exit the general event handler.[43]

18.6.11 Procedure for Handling the MFC SPU Command Queue Available Event

The MFC SPU Command Queue Available event is raised when the channel count for MFC Class ID and Command Opcode (`MFC_Cmd`) changes from ‘0’ (full) to non-zero (not-full). The occurrence of the event means that the SPU can write `MFC_Cmd` to enqueue a command at least once without stalling.

When the MFC SPU Command Queue Available event handler executes, the MFC SPU Command Queue may have more than one empty entry. Since the event is only raised when the `MFC_Cmd` channel count changes from ‘0’ to non-zero, the handler can improve efficiency by enqueuing commands until the command queue is full. Thus, software may implement a software queue of MFC commands that is shared by the handler and the main application code.

The procedure for handling the MFC SPU Command Queue Available Event is:

1. Perform a read channel (**rdch**) instruction to the SPU Read Event Mask Channel and save the data in the “mask”.[44]
2. Mask the event by issuing a write channel (**wrch**) instruction to the SPU Write Event Mask Channel with `SPU_WrEventMask[Qv]` set to ‘0’.[44]
3. Acknowledge the event by performing a write channel (**wrch**) instruction to the SPU Write Event Acknowledgment Channel with `SPU_WrEventAck[Qv]` set to ‘1’.
4. Obtain the channel count by issuing a read channel count (**rchcnt**) instruction to the MFC Command Opcode Channel (`MFC_Cmd`).
5. If the channel count is ‘0’, skip to step 8.
6. Enqueue a DMA command to the MFC command queue.
7. If more commands are left to queue, return to step 3.
8. Exit the SPU Command Queue handler.
9. Restore the “mask” by issuing a write channel (**wrch**) instruction to the SPU Write Event Mask Channel with `SPU_WrEventMask[Qv]` set to ‘1’.[44]
10. Exit the general event handler.[44]

---

44. When multiple events are enabled, a common handler should be used to save the current event mask, mask all events that are to be handled in one channel write, and acknowledge all events that are to be handled in a single channel write. Then each event specific handler should be invoked to handle the event. The common handler should then restore the current event mask from the saved value and exit. This technique minimizes the generation of spurious events.

18.6.12 Procedure for Handling the DMA List Command Stall-and-Notify Event

The DMA List Command Stall-and-Notify event is raised when the DMA hardware completes an element of a DMA list command list and the element has the stall-and-notify flag set to ‘1’ (the stall-and-notify flag is bit 0 (the MSb) of the transfer-size word of the element); when the channel count for MFC Read List Stall-and-Notify Tag Status (`MFC_RdListStallStat`) changes from ‘0’ to ‘1’, the event is raised.

At this point, the DMA hardware suspends processing of the DMA list and raises the event to give SPE software an opportunity to modify the characteristics (transfer size, starting effective address, stall-and-notify flag) of the remaining elements in the list. See *Section 19 DMA Transfers and Inter-Processor Communication* on page 505 for an example of how the stall-and-notify event can be used.

The DMA hardware in the MFC is designed to execute DMA operations with maximum performance. To this end, the DMA hardware can execute the DMA commands from its queues out of order. When the stall-and-notify event is raised, SPE software can query `MFC_RdListStallStat` to find out which tag groups have a stalled DMA list command, but the channel interface provides no information about which DMA list command in the tag group has stalled and no indication of which element in the DMA list command has stalled.

When a DMA list contains multiple elements with the stall-and-notify flag set to ‘1’ or when a tag group has multiple DMA list commands queued that have elements with the stall-and-notify flag set, application software must track stall-and-notify events explicitly. Software can maintain a stall counter for each tag group with elements that can stall, but in addition, the DMA list commands within each such tag group must be explicitly ordered with tag-specific fences or barriers or with the command barrier. Without enforced ordering, DMA hardware can execute DMA commands out of order; if hardware executes commands out of order, software is unable to identify which DMA list command within the tag group is causing a given stall-and-notify event.

When the DMA List Command Stall-and-Notify event handler executes, `MFC_RdListStallStat` may have more than bit set to ‘1’, indicating more than one group with a stalled DMA command. If the handler does not acknowledge and resume all stalled tag groups in response to the event by writing `MFC_WrListStallAck` for each group, the un-acknowledged stalled groups will not cause a subsequent stall-and-notify event and they will remain in the stalled state. Thus, the handler should process and acknowledge all stalled groups reported in `MFC_RdListStallStat`.

Software can adjust the characteristics of a stalled DMA list command in a tag group by changing list-element addresses and transfer sizes for the list elements beyond the element that caused the stall. If application software determines that some list elements should be skipped, software can simply set the list-element transfer sizes to ‘0’. However, the number of list elements in a queued DMA list command cannot be changed.

The procedure for handling the DMA List Command Stall-and-Notify Event is:

1. Perform a read channel (**rdch**) instruction to the SPU Read Event Mask Channel and save the data in the “mask”.[45]

45. When multiple events are enabled, a common handler should be used to save the current event mask, mask all events that are to be handled in one channel write, and acknowledge all events that are to be handled in a single channel write. Then each event specific handler should be invoked to handle the event. The common handler should then restore the current event mask from the saved value and exit. This technique minimizes the generation of spurious events.

2. Mask the event by issuing a write channel instruction to the SPU Write Event Mask Channel with `SPU_WrEventMask[Sn]` set to '0'.[45]
3. Acknowledge the event by performing a write channel (**wrch**) instruction to the SPU Write Event Acknowledgment Channel with `SPU_WrEventAck[Sn]` set to '1'.[45]
4. Perform a read channel (**rdch**) instruction to the MFC Read List Stall-and-Notify Tag Status Channel `MFC_RdListStallStat[gn]`.
5. Use this information to determine which tag group or tag groups have a DMA List Element in the Stall and Notify state.
6. Perform the application-specific action with respect to each tag group having a stalled DMA list element.

   Note: *If a DMA list contains multiple list elements having the Stall and Notify flag set, or if a Tag Group has multiple DMA list commands queued with elements having the Stall and Notify flag set, it is essential for the application software to initialize to 0 a tag-group specific stall counter before the DMA list commands are queued for the tag group. In addition, if multiple DMA list commands are queued for a tag group with Stall and Notify elements, ordering must be enforced with tag-specific fences, barriers, or the command barrier. Each time a Stall and Notify status is indicated for a tag group, the corresponding counter should be incremented. Application software can then use this counter to determine at what point in the list the stall has occurred. Application software uses stall and notify to update list element addresses and transfer sizes that follow the list element that has stalled due to dynamically changing conditions. List elements after the stalled list element can be skipped by setting their transfer sizes to 0. However the number of list elements in a queued DMA list command cannot be changed.*
7. Acknowledge and resume each stalled DMA list command by issuing a write channel (**wrch**) instruction to the MFC Write List Stall-and-Notify Tag Acknowledgment Channel (`MFC_WrListStallAck[MFC Tag]`) where the supplied MFC Tag is the encoded Tag ID of the tag group to be resumed.
8. Exit the DMA List Stall and Notify handler.

   Note: *If application software does not acknowledge all stalled tag groups indicated in the MFC_RdListStallStat[gn] channel, a second stall and notify event does not occur for the unacknowledged tag group.*
9. Restore the "mask" by issuing a write channel (**wrch**) instruction to the SPU Write Event Mask Channel with `SPU_WrEventMask[Sn]` set to '1'.[46]
10. Exit the general event handler.[45]

---

46. If multiple events are enabled, a common handler should be used to save the current event mask, mask all events that are to be handled in one channel write, and acknowledge all events that are to be handled in a single channel write. Then each event specific handler should be invoked to handle the event. The common handler should then restore the current event mask from the saved value and exit. This technique minimizes the generation of spurious events.

18.6.13 Procedure for Handling the Tag-Group Status Update Event

The Tag-Group Status Update event is raised when a tag group or groups have completed and resulted in an update of the MFC Read Tag-Group Status (`MFC_RdTagStat`); when the channel count for `MFC_RdTagStat` changes from ‘0’ to ‘1’, the event is raised, and software can read `MFC_RdTagStat` without stalling.

The specific meaning of this event is determined by the values that software writes to the MFC Write Tag Group Query Mask (`MFC_WrTagMask`) and MFC Write Tag Status Update Request (`MFC_WrTagUpdate`). Software writes a mask to `MFC_WrTagMask` to track completion status for the groups with corresponding bits set to ‘1’ in the mask. Then, software writes a request type to `MFC_WrTagUpdate`. If the request type is “any,” then when any of the groups named in the mask complete, `MFC_RdTagStat` is updated and the event is raised. If the request type is “all,” then when all the groups named in the mask complete, `MFC_RdTagStat` is updated and the event is raised. If the request type is “immediate,” then `MFC_RdTagStat` is updated immediately and the event is raised.

The steps in the event-handling protocol are shown below:

1. Read channel count (**rchcnt**) for `MFC_RdTagStat`.
2. If channel count is ‘0’ then handler done.
3. Read `MFC_RdTagStat`.
4. Perform application-specific processing for each completed tag group.

18.7 Developing a Basic Interrupt Handler

Since SPE hardware provides only basic interrupt support, SPE software is responsible for defining most aspects of interrupt handling. Most programmers will want to use high-level languages and tools as much as possible, so the calling conventions and stack management defined by the high-level language development tools should guide the development of the interrupt handling code.

Proper handling of external events on the SPE involves a certain degree of complexity. From a software engineering perspective, this complexity is best managed when a set of system utility functions are provided. To this end, the following items are considered necessary to promote the use of external events (and asynchronous interrupts) on the SPE:

- A set of utilities that safely mask, unmask, and acknowledge a set of external events.
- A FLIH that saves and restores resources on entry and exit, and that supports SLIH’s that conform to the ABI of the SPE program. A set of utilities for registering SLIH’s should also be included.
- A set of utilities to query, enable, disable, and save interrupt state, thus supporting nested interrupts and SPE critical sections.[47]

Much of this could be absorbed into the default SPE entry function (`crt0.o`), or efficiently implemented as macros or inline assembly functions, with little or no cost to the general SPE program’s LS footprint.

47. The Linux kernel supports interfaces such as `in_interrupt()`, `local_irq_enable()`, `local_irq_disable()`, `local_irq_save()`, `local_irq_restore()`, and so forth, for these purposes.

The focus of the discussion in this section is on developing an interrupt handler that is compatible with function-calling conventions and stack management, as defined in the *SPU Application Binary Interface Specification*. Compatibility with the ABI allows the interrupt handler(s) to be at least partly written in a high-level language.

18.7.1 Basic Interrupt Protocol Features and Design

The example interrupt protocol shown here will consist of a small first-level interrupt handler (FLIH) coded in assembler language and a small routine for registering second-level interrupt handlers (SLIH) coded entirely in C. The FLIH respects the register-use and stack-management requirements of the SPE ABI, and the SLIH registration code is written in C. Thus, SPE application developers can write SLIH event handlers entirely in C and register these SLIH event handlers with a simple C function call. The example shows that it is possible to have an SPU interrupt mechanism that is easy to use, although a handler written entirely in assembly code would be more efficient.

The example concludes with application code that registers a decrementer event handler and counts ten decrementer events.

18.7.2 FLIH Design

The FLIH code resides at address zero in the LS, so it runs every time the SPE takes an interrupt. The tasks of the FLIH are:

- Preserve the interrupted application stack.
- Invoke the SLIH for the pending event.
- Restore the application stack and return to the interrupted application.

The FLIH assembler code is shown below:

```
.set STACK_SKIP, 125                # max. size needed to protect unallocated stack space
        .extern spu_slih_handlers
        .text
        .align 3
        .global spu_flih
spu_flih:
        stqd         $SP, -(STACK_SKIP+81)*16($SP)    # save back-chain pointer
        stqd         $0, -(STACK_SKIP+80)*16($SP)     # save volatile registers 0, 2-79
        stqd         $2, -(STACK_SKIP+ 2)*16($SP)
        stqd         $3, -(STACK_SKIP+ 3)*16($SP)
        stqd         $4, -(STACK_SKIP+ 4)*16($SP)
        stqd         $5, -(STACK_SKIP+ 5)*16($SP)
        stqd         $6, -(STACK_SKIP+ 6)*16($SP)
        stqd         $7, -(STACK_SKIP+ 7)*16($SP)
        stqd         $8, -(STACK_SKIP+ 8)*16($SP)
        stqd         $9, -(STACK_SKIP+ 9)*16($SP)
        stqd         $10, -(STACK_SKIP+ 10)*16($SP)
                . . .                              # omitted regs 11-75 for brevity
        stqd         $76, -(STACK_SKIP+76)*16($SP)
        stqd         $77, -(STACK_SKIP+77)*16($SP)
```

```
        stqd        $78, -(STACK_SKIP+78)*16($SP)
        stqd        $79, -(STACK_SKIP+79)*16($SP)
        rdch        $6, $SPU_RdEventMask
        stqd        $6, -(STACK_SKIP+1)*16($SP)     # save event mask on stack
        rdch        $3, $SPU_RdEventStat
        andc        $7, $6, $3                      # clear current pending event bits in mask
        wrch        $SPU_WrEventMask, $7            # disable current pending events
        wrch        $SPU_WrEventAck, $3             # acknowledge current pending events
        il          $2, -(STACK_SKIP+81)*16         # stack frame size
        a           $SP, $SP, $2                    # instantiate flih stack frame
next_event:
        clz         $4, $3                          # determine next (left-most) event
        ila        $5, spu_slih_handlers            # address of slih function table
        shli       $4, $4, 2                        # make event number a word offset
        lqx        $5, $4, $5                       # load four slih function pointers
        rotqby     $5, $5, $4                       # rotate slih pointer to preferred slot
        bisl       $0, $5                           # branch-and-link to slih function
        brnz       $3, next_event                   # more events? ($3 is slih return value)
        lqd        $6, 80*16($SP)
        wrch        $SPU_WrEventMask, $6            # re-establish previous event mask
        lqd        $79, 2*16($SP)                   # restore volatile registers 79-2, 0
        lqd        $78, 3*16($SP)
        lqd        $77, 4*16($SP)
        lqd        $76, 5*16($SP)
                . . .
        lqd        $10, 71*16($SP)
        lqd        $9, 72*16($SP)
        lqd        $8, 73*16($SP)
        lqd        $7, 74*16($SP)
        lqd        $6, 75*16($SP)
        lqd        $5, 76*16($SP)
        lqd        $4, 77*16($SP)
        lqd        $3, 78*16($SP)
        lqd        $2, 79*16($SP)
        lqd        $0, 1*16($SP)
        lqd        $SP, 0*16($SP)                # restore stack pointer from back chain ptr
        irete                                    # Return through SRR0, re-enable interrupts
```

18.7.2.1 *Preserve the Interrupted Application Stack*

The first section of FLIH code allocates an ABI-compliant stack frame. Since the FLIH cannot know the exact context of the application code that has been interrupted, it must assume the worst-case situation, which happens when two conditions are true:

- A leaf function (a function that makes no function calls) has been interrupted.
- The leaf function uses space in its stack frame without actually allocating the frame.

It is legal for a leaf function to use stack space below its caller's stack frame without actually decrementing the stack pointer to allocate a frame. Thus, to be compatible with any interrupt scenario, the FLIH must protect space on the stack equivalent to the maximum size stack frame and preserve registers so that the code for the SLIH, which is written in an ABI-compliant high-level language, can freely use the processor registers and stack according to the ABI rules.

The example FLIH begins by storing the previous stack frame's back-chain pointer at the end of the new stack frame. Next, it saves the return address from R0 (LR, the link register) to the correct position in the new stack frame, just above the back-chain pointer. Next, the FLIH saves volatile registers including the registers that might contain parameters for the interrupted function. Then the FLIH sets the SP to the bottom of the new stack frame. At this point, the stack is ready to support calling ABI-compliant functions.

The FLIH also follows the recommended policy of acknowledging all the current pending events before the first event is processed.

18.7.2.2 *Invoke SLIH for Pending Event*

The middle section of the FLIH gets the pending event status, determines the left-most event, loads, aligns the function pointer for the event's corresponding SLIH, and executes a branch-and-link to the SLIH function. Since an ABI-compliant stack frame has been allocated and volatile registers have been saved, the SLIH can be a high-level-language function.

Note that the event status is in R3, which is the first parameter register according to the ABI protocol. Thus, the effect of the FLIH call to the SLIH is equivalent to the C function call:

```
specific_slih (event_status);
```

When the SLIH returns, the FLIH expects the SLIH to have cleared its corresponding event bit in the `event_status` and to have passed back this updated status as the return value; the return value is passed back in R3 according to the SPE ABI. The FLIH uses the return value to determine the next left-most pending event and loops to call the next event's corresponding SLIH.

18.7.2.3 *Restore Application Stack and Return to Interrupted Application*

When all pending events have been handled, the FLIH restores the context of the interrupted code by restoring the saved volatile registers and unwinding the stack. The SP is reset to point to the stack frame of the interrupted code, and the FLIH exits with an **irete** instruction to re-enabled interrupts.

Note that the FLIH and all the SLIH functions are executed with interrupts disabled.

18.7.3 SLIH Design and Registering SLIH Functions

SLIH functions are expected to be written as part of an application that is coded in a high-level language; here, we use the C language. The code to register the SLIH functions is also written in C. Shown below is the code for the default SLIH function, the code for the initialization of the SLIH function table, and the function that application code uses to register SLIH functions.

```
#define SPU_EVENT_ID(_mask)(spu_extract(spu_cntlz(spu_promote(_mask, 0)), 0))
```

```
static unsigned int spu_default_slih (unsigned int events)
{
    unsigned int mse;

    mse = 0x80000000 >> SPU_EVENT_ID (events); /* get my (left-most) event number */

    return (events & ~mse);                    /* return updated event bits */
}

spu_slih_func  spu_slih_handlers[33] __attribute__ ((aligned (16))) = {
    spu_default_slih, /* event bits 0 through 18: RESERVED, use default slih */
    spu_default_slih,
    spu_default_slih,
    spu_default_slih,
    spu_default_slih,
    spu_default_slih,
    spu_default_slih,
    spu_default_slih,

    spu_default_slih,
    spu_default_slih,
    spu_default_slih,
    spu_default_slih,
    spu_default_slih,
    spu_default_slih,
    spu_default_slih,
    spu_default_slih,

    spu_default_slih,
    spu_default_slih,
    spu_default_slih,
    spu_default_slih, /* event bit 19: Multisource Sync */
    spu_default_slih, /* event bit 20: Privilege attention */
    spu_default_slih, /* event bit 21: Lock-line reservation-lost */
    spu_default_slih, /* event bit 22: Signal-notification 1 */
    spu_default_slih, /* event bit 23: Signal-notification 2 */

    spu_default_slih, /* event bit 24: SPU Write Outbound Mailbox available */
    spu_default_slih, /* event bit 25: SPU Write Outbound Interrupt Mailbox avail. */
    spu_default_slih, /* event bit 26: Decrementer */
    spu_default_slih, /* event bit 27: SPE mailbox */
    spu_default_slih, /* event bit 28: DMA-queue */
    spu_default_slih, /* reserved */
    spu_default_slih, /* event bit 30: DMA list command stall-and-notify */
    spu_default_slih, /* event bit 31: SPE tag-status update */

    spu_default_slih, /* extra slih table entry for phantom events */
};
```

```
void spu_slih_reg(unsigned int mask, spu_slih_func func)
{
    unsigned int id;

    while (mask)
    {
        id = SPU_EVENT_ID (mask);      /* get next (left-most) event */
        spu_slih_handlers[id] = func;  /* put function pointer into slih table */
        mask &= ~(0x80000000 >> id);   /* clear that event bit */
    }
}
```

18.7.3.1 *Default SLIH Function*

The default SLIH function, `spu_default_slih`, simply satisfies the FLIH protocol that requires each SLIH to clear its corresponding event bit in the status. The event bit is cleared the in the status and the updated status is returned to the caller (the FLIH), but the default SLIH takes no other action on behalf of the event.

18.7.3.2 *Initialization of SLIH Function Table*

The SLIH function table has 32 entries for SLIH functions that correspond to event bits 0 through 31. Each entry in the table is initialized to the default SLIH function, `spu_default_slih()`.

The function table has a 33rd entry, also initialized to `spu_default_slih()`, that will be accessed when an interrupt is taken with no event bits set in the SPU Read Event Status channel. Such an interrupt is called a “phantom event” interrupt. In this case, the count-leading-zeros instruction (**clz**) in the FLIH code will return the value 32, which will cause the 33rd entry in the table to be used as the function pointer. For a phantom event interrupt, the correct action is to read the SPU Read Event Status channel, which clears the phantom event, and return from the interrupt. If a phantom event occurs, the code shown above handles it properly.

18.7.3.3 *Registering an Application Function in SLIH Table*

The last function in the code above, `spu_slih_reg()`, is used by an application to register (install) its SLIH functions. Normally, an application will have one function for each event of interest and will call `spu_slih_reg()` with a single-bit mask to install each SLIH function. This `spu_slih_reg()` function is written, however, to allow a single call to register the same function for multiple events. In the extreme, an application could register a single SLIH for all events of interest; this allows the application’s SLIH to supersede the FLIH dispatch and move most aspects of event processing into application code.

18.7.4 Example Application Code

With the interrupt-handling infrastructure described above, only a small amount of programming is needed to enable and process asynchronous events, and no assembler programming is required.

Shown below is a simple application program that sets up an event handler for decrementer events (a decrementer event occurs when the SPU decrementer count makes the transition from '0' to '-1', see *Section 18.10.1 SPU Decrementer Event* on page 497) and waits until ten such events occur before printing an appropriate message.

```
#include <stdio.h>
#include <spu_mfcio.h>
#define DECR_COUNT        10000    /* count 10000 timebase ticks per decrementer interrupt */

volatile unsigned int event_count = 0;

unsigned int decr_handler (unsigned int status)
{
        event_count++;                            /* bump global event count */
        spu_writech (SPU_WrDec, DECR_COUNT);      /* reset decrementer */
        return (status & ~MFC_DECREMENTER_EVENT); /* return updated event bits */
}

int main (void)
{
        spu_slih_reg (MFC_DECREMENTER_EVENT, decr_handler);    /* register the handler */
        spu_writech (SPU_WrEventMask, MFC_DECREMENTER_EVENT);  /* enable decr. event */
        spu_writech (SPU_WrDec, DECR_COUNT);                   /* init decrementer */
        spu_ienable ();                                        /* enable interrupts */
        while (event_count < 10);          /* busy-wait until 10 decrementer events happen */
        printf ("Test successfully completed.\n");
        return (0);
}
```

18.7.4.1 *Decrementer Event Handler*

The decrementer event handler is based on the default SLIH shown previously. In addition to clearing the event bit in the status, the handler increments the global `event_count` variable, re-initializes the decrementer to the start value, and returns the event status with the decrementer-event bit cleared.

18.7.4.2 *Application Code*

The application code for this example demonstrates how simple it is for SPU application programs to setup and use asynchronous events. The main program first registers the SLIH `decr_handler()` in the SLIH function table by calling the `spu_slih_reg()` function defined above. Next, the decrementer event is enabled by writing a mask to the SPU Write Event Mask channel. Next, the decrementer is initialized to the start value `DECR_COUNT` and interrupts are enabled. The empty while loop implements a busy-wait until ten decrementer events occur. Finally, the application prints a success message and exits.

In the application, only four lines of code are required to setup interrupt handling including the line to initialize the decrementer starting count. The decrementer SLIH also needs only four lines of code, including re-initializing the starting count and updating the global event counter. A more

sophisticated application that uses the decrementer to help implement execution profiling of a running application will be discussed below in *Section 18.10.1 SPU Decrementer Event* on page 497.

18.8 Nested Interrupt Handling

The interrupt handling design described above is simple and clean but has the limitation that all pending events must be processed before another event interrupt can be taken. When many events are handled in one invocation of the FLIH, this limitation can lead to long latency between an event occurring and its handler running. Nested interrupt handling removes this limitation and is beneficial when an application must guarantee minimum delay between the occurrence of a high-priority event and handling the event.

For example, it is best to handle DMA-queue interrupts as soon as they occur. CBE system throughput is maximized when all processing elements, including the DMA engines in the MFCs, are active simultaneously. If the mailbox event handler requires lengthy processing and runs with interrupts disabled, a DMA-queue event that occurs shortly after the mailbox handler starts will not be recognized immediately, and, consequently, overall system throughput may be reduced by an idle DMA engine. If nested interrupt handling is implemented, the mailbox handler can be interrupted, which allows immediate handling of a DMA-queue event to minimize DMA idle time.

18.8.1 Nested Handler Design

In the basic design described above, each SLIH was written as a high-level-language function, which means SLIH functions need no modification to support interruptibility. The FLIH code, however, does need some changes to support interruption of SLIH functions.

The implication of interrupting a SLIH function is that the FLIH can run again before a previous FLIH execution runs to completion. The FLIH itself need not be interruptible, but it must save some extra machine context information and re-enable interrupts before transferring control to an interruptible SLIH function. Since the FLIH transfers control to a SLIH before the FLIH finishes all its work, the FLIH must preserve its own context in addition to the context of the interrupted program.

The example code uses a fixed, bit-ordered priority. The code could be modified to handle interrupts in a different priority by bit-sizzling the interrupt events.

18.8.2 FLIH Design for Nested Interrupts

The nested FLIH assembler code is shown below with new or changed lines of code shown in bold-face type:

```
        .set STACK_SKIP, 125 # max. size needed to protect unallocated stack space
        .extern spu_slih_handlers
        .text
        .align 3
        .global spu_flih
spu_flih:
        stqd         $SP, -(STACK_SKIP+82)*16($SP)   # save back-chain pointer
        stqd         $0, -(STACK_SKIP+81)*16($SP)    # save volatile registers 0, 2-79
```

```
        stqd        $2, -(STACK_SKIP+ 2)*16($SP)
        stqd        $3, -(STACK_SKIP+ 3)*16($SP)
        stqd        $4, -(STACK_SKIP+ 4)*16($SP)
        stqd        $5, -(STACK_SKIP+ 5)*16($SP)
        stqd        $6, -(STACK_SKIP+ 6)*16($SP)
        stqd        $7, -(STACK_SKIP+ 7)*16($SP)
        stqd        $8, -(STACK_SKIP+ 8)*16($SP)
        stqd        $9, -(STACK_SKIP+ 9)*16($SP)
        stqd        $10, -(STACK_SKIP+ 10)*16($SP)
                . . .                           # omitted regs 11-75 for brevity
        stqd        $76, -(STACK_SKIP+76)*16($SP)
        stqd        $77, -(STACK_SKIP+77)*16($SP)
        stqd        $78, -(STACK_SKIP+78)*16($SP)
        stqd        $79, -(STACK_SKIP+79)*16($SP)
        rdch        $3, $SPU_RdSRR0             # get interrupt return address
        stqd        $3, -(STACK_SKIP+80)*16($SP)  # save int. ret. addr. above link reg.
        rdch        $6, $SPU_RdEventMask
        stqd        $6, -(STACK_SKIP+1)*16($SP)   # save event mask on stack
        rdch        $3, $SPU_RdEventStat
        andc        $7, $6, $3                  # clear current pending event bits in mask
        wrch        $SPU_WrEventMask, $7        # disable current pending events
        wrch        $SPU_WrEventAck, $3         # acknowledge current pending events
        il          $2, -(STACK_SKIP+82)*16     # get stack frame size
        a           $SP, $SP, $2                # instantiate flih stack frame
next_event:
        clz         $4, $3                      # determine next (left-most) event
        ila         $5, spu_slih_handlers       # address of slih function table
        shli        $4, $4, 2                   # make event number a word offset
        lqx         $5, $4, $5                  # load four slih function pointers
        rotqby $5, $5, $4                       # rotate slih pointer to preferred slot
        bisle       $0, $5                      # call slih with interrupts enabled
        brnz        $3, next_event              # more events? ($3 is slih return value)
        ila         $0, next_inst               # disable interrupts before restoreing SRR0
        bid         $0
next_inst:
        lqd         $6, 80*16($SP)
        wrch        $SPU_WrEventMask, $6        # re-establish previous event mask
        lqd         $0, 1*16($SP)
        wrch        $SPU_WrSRR0, $0             # restore interrupt return address

        lqd         $79, 2*16($SP)              # restore volatile registers 79-2, 0
        lqd         $78, 3*16($SP)
        lqd         $77, 4*16($SP)
        lqd         $76, 5*16($SP)
                . . .
        lqd         $10, 71*16($SP)
        lqd         $9, 72*16($SP)
        lqd         $8, 73*16($SP)
        lqd         $7, 74*16($SP)
        lqd         $6, 75*16($SP)
```

```
lqd         $5, 76*16($SP)
lqd         $4, 77*16($SP)
lqd         $3, 78*16($SP)
lqd         $2, 79*16($SP)
lqd         $0, 1*16($SP)
lqd         $SP, 0*16($SP)              # restore stack pointer from back chain ptr
syncc                                   # force channel write to complete
irete                                   # Return through SRR0, re-enable interrupts
```

The new FLIH, which allows SLIH functions to be interrupted, has only a few minor changes. First, the stack frame that is allocated to protect the context of the interrupted code has been expanded by one quadword. The extra quadword is required to hold the interrupt return address and maintain quadword alignment of the stack. The interrupt return address will be saved on the stack just above the saved R0 (LR), so the offsets for the back pointer and link register in the stack have been adjusted down by one quadword. The interrupt return address is moved from the Save-Restore Register 0 channel into R0 and then stored onto the stack.

The indirect-branch instruction that calls SLIH functions, **bisl**, has been changed into the version with the `E` feature bit set, **bisle**. Thus, when the first instruction of the SLIH function executes, interrupts will be enabled.

When the SLIH function returns, the FLIH code, as before, looks for more pending events as indicated by bits set to ‘1’ in the updated SPU Read Event Status channel that is passed back from the SLIH. When all enabled pending events are serviced by calls to SLIH functions, the FLIH context-restore code is executed. The context-restore code is unchanged except for the offsets in the instructions that restore R0 (LR) and SP, the two new instructions that restore the Save-Restore Register 0 value, and the **syncc** instruction that forces the SPU Write State Save-and-Restore channel write to complete before the **irete** executes. Prior to restoring the Save-Restore Register and stack pointer, the interrupts must be disabled. They are re-enabled at end of the FLIH by executing a **irete** instruction.

18.9 Using a Dedicated Interrupt Stack

The previous examples of interrupt handling code have used the application stack for temporary storage space and to support calling SLIH functions. However, using the application stack for interrupt handling creates some problems.

One problem is the fact that the FLIH must allocate a very large frame to account for the worst-case situation: interrupting a leaf procedure that is using a maximum-sized stack frame without actually allocating it. The FLIH must allocate more than 125 quadwords—2000 bytes—on the stack even though only a small fraction of that space is likely ever needed.

Another problem is switching contexts when the run-time environment supports multiple processes or multiple threads. When an interrupt handler can be the cause of a process or thread switch, using a separate, dedicated interrupt stack can simplify preserving the context of the suspended thread and restoring the context of the activated thread.

The FLIH code shown below implements a separate interrupt stack; this dedicated stack is used by the FLIH code and SLIH functions. The major changes in the FLIH are to save the application stack pointer and initialize the SP register to the top of the dedicated interrupt stack. Also, this FLIH allocates a much smaller stack frame because the frame need only have space for the volatile registers that must be preserved across calls to SLIH functions.

When the FLIH is finished, it restores the application stack pointer and returns from the interrupt. In a more complete interrupt handler—one that supports thread switching—the application stack pointer would likely be saved in a thread context structure to support a possible interrupt-induced thread switch.

```
.set INTERRUPT_STACK_SIZE, 2048

        .extern spu_slih_handlers
        .text
        .align 3
        .global spu_flih
spu_flih:
        stqr $SP, main_stack_ptr
        ila $SP, interrupt_stack-81
        stqd $0, 1*16($SP)                # save volatile registers 0, 2-79
        stqd $SP, 0*16($SP)               # save back chain pointer
        stqd $2, 80*16($SP)
        stqd $3, 79*16($SP)
        stqd $4, 78*16($SP)
        stqd $5, 77*16($SP)
        stqd $6, 76*16($SP)
        stqd $7, 75*16($SP)
        stqd $8, 74*16($SP)
        stqd $9, 73*16($SP)
        stqd $10, 72*16($SP)
        . . .                          # omitted regs 11-75 for brevity
        stqd $76, 6*16($SP)
        stqd $77, 5*16($SP)
        stqd $78, 4*16($SP)
        stqd $79, 3*16($SP)
        rdch $6, $SPU_RdEventMask
        stqd $6, 2*16($SP)            # save event mask on stack
        rdch $3, $SPU_RdEventStat
        andc $7, $6, $3               # clear current pending event bits in mask
        wrch $SPU_WrEventMask, $7     # disable current pending events
        wrch $SPU_WrEventAck, $3      # acknowledge current pending events
next_event:
        clz $4, $3                    # determine next (left-most) event
        ila $5, spu_slih_handlers     # address of slih function table
        shli $4, $4, 2                # make event number a word offset
        lqx $5, $4, $5                # load four slih function pointers
        rotqby $5, $5, $4             # rot slih function pointer to preferred slot
        lqd $4, 2*16($SP)             # pass interrupt return address to slih
        bisl $0, $5                   # call slih
        brnz $3, next_event           # more events? ($3 is slih return value)
```

```
        lqd $6, 2*16($SP)
        wrch $SPU_WrEventMask, $6     # re-establish previous event mask
        lqd $79, 3*16($SP)            # restore volatile registers 79-2, 0
        lqd $78, 4*16($SP)
        lqd $77, 5*16($SP)
        lqd $76, 6*16($SP)
                . . .
        lqd $10, 72*16($SP)
        lqd $9, 73*16($SP)
        lqd $8, 74*16($SP)
        lqd $7, 75*16($SP)
        lqd $6, 76*16($SP)
        lqd $5, 77*16($SP)
        lqd $4, 78*16($SP)
        lqd $3, 79*16($SP)
        lqd $2, 80*16($SP)
        lqd $0, 1*16($SP)
        lqr $SP, main_stack_ptr      # restore application stack pointer
        irete                        # return through SRR0 and re-enable interrupts

        .align 4
        .skip INTERRUPT_STACK_SIZE
interrupt_stack:                     # top of private interrupt stack
main_stack_ptr:                      # Place to save main runtime stack pointer
        .skip 16
```

18.10 Sample Applications

This section considers a few basic examples of how external events might be used by an SPE program.

A preferred implementation would enable asynchronous interrupts and process specific conditions with SLIH's, which are typically written in a higher-level language like C. Of course **bisled** polling or individual event polling could be used instead, provided that the event status is checked at appropriate intervals.

18.10.1 SPU Decrementer Event

The decrementer and the SPU Decrementer event have many potential uses, including the watchdog timer function discussed below in *Section 18.10.3.1 Watchdog Timer* on page 499. A few of these are described in the next few sections.

Each SPE controls a 32-bit decrementer. The decrementer value counts down by '1' for every cycle of the PPE time base (see *Section 13.2.3 Time-Base Frequency* on page 377). If the decrementer event is enabled (the `Tm` bit is set in the SPU Read Event Mask), a decrementer event occurs when the value in the decrementer makes the transition from '0' to '-1' (that is, the most-significant bit changes from '0' to '1').

Since the frequency of the time base is a function of the system, SPE software can assume that a specific decrementer value corresponds to a fixed time interval, regardless of the CBE implementation. Thus, software can use the decrementer for time-based scheduling, performance monitoring, and so forth.

If the value loaded into the decrementer causes a change from ‘0’ to ‘1’ in the MSb, an event is signaled immediately. Setting the decrementer to a value of ‘0’ results in an event after a single decrementer interval.

18.10.1.1 *Execution Profiling or Hot-Spot Analysis*

Often, compute-intensive inner loops offer the best opportunities to improve program performance. Programmers can usually identify inner loops by inspection, yet programmer intuition about where to apply optimization effort is subjective and sometimes wrong.

Execution profiling provides objective data about where a program’s run time is actually being spent. High-level-language development systems usually have a way to build a version of any program that will gather execution profile information. This type of profiling works well for non-real-time applications, but it has disadvantages due to the extra code that is inserted by the development system to gather the profile data; this extra code changes the real-time behavior of the program by adding to execution time and changing the memory addresses of processor instructions. As a result of the extra profile-gathering code, cache residency and branch alignments may change in ways that affect execution characteristics.

The disadvantages of adding code for profiling can be reduced by gathering profile data through periodic interrupts. When a time interval expires, an interrupt is taken, and the interrupt handler makes note of where the program was executing by incrementing a count associated with the range of PC addresses that contains the interrupt PC. The handler maintains an array of counts such that the product of the size of the array and the range of PC addresses per count covers the application code of interest. If the interrupt interval is short enough and the application run time long enough, statistical sampling theory holds that the resulting array of counts accurately represents where the application is spending its execution time. The array of counts can be analyzed when the application terminates or even while the application is running by correlating the PC address ranges with high-level-language statements and procedure names.

18.10.1.2 *Garbage Collection*

A handler for the decrementer event could be used to intermittently run a garbage collector for languages supporting such constructs, such as C++ or Java.

18.10.1.3 *Microthreads*

A user-level threading (microthread) environment for the SPE could use the decrementer event to time slice between tasks that share the same SPE.

18.10.2 Tag-Group Status Update Event

The Tag-Group Status Update event becomes pending when all the MFC commands are complete in at least one unmasked tag group. Since a tag group by definition contains multiple MFC commands, completion of all commands may take a significant amount of time; instead of wasting time waiting for tag-group completion, it makes sense to have the SPU work on other tasks while the MFC autonomously executes the commands in the tag group.

To support such concurrency, SPE software can include an event handler that responds to the Tag-Group Status Update Event. The handler can set a buffer-complete flag for each tag group reported in the value read from the MFC Read Tag-Group Status channel. Software can set up a data structure with status bits for each buffer that is currently being processed by the MFC. The structure could have multiple 32-bit status, if necessary, where the assignment of buffer flag bits in the 32-bit words matches the tag-group bit assignments in the 32-bit word of the MFC Read Tag-Group Status channel. When the Tag-Group Status Update event is recognized, the handler can efficiently set and clear flags, such as buffer-complete or buffer-busy flags, with bit-wise logical operations.

18.10.3 DMA List Command Stall-and-Notify Event

The purpose of the DMA List Command Stall-and-Notify event is to allow software to set up and start a sequence, or list, of DMA transfers even when the precise characteristics of all list-element transfers is not known at set-up time. By setting the stall-and-notify flag in one or more list elements, the DMA list command can be queued, and the MFC can immediately begin processing some of the DMA transfers.

After the DMA transfer for the first list element in the list with the stall-and-notify flag is completed, the MFC suspends list processing and raises the DMA List Command Stall-and-Notify event. In response to the event, software can re-evaluate conditions affecting the DMA list transfer and adjust the characteristics—transfer size and effective address—of one or more of the remaining list elements. When adjustments are complete, software restarts the MFC processing of the list by acknowledging the event.

When software adjusts list-element characteristics, it can set the stall-and-notify flag in subsequent elements, and the process of responding the event, re-evaluating conditions, and adjusting list-element characteristics can be repeated.

Two example applications of this event-handling procedure are presented below.

18.10.3.1 *Watchdog Timer*

In real-time applications, such as computing frames in a live-action 3D game, software must respond to strict deadlines. Software may be unable to complete all requested processing in a real-time interval due to unanticipated input data or environment complexity, unusually high input data rate, and so forth. In a 3D game, if the current frame cannot be completed in its allotted time, it may be acceptable to have software drop the current frame or render a low-quality version and begin rendering the next frame.

SPE software can combine the DMA List Command Stall-and-Notify event with a real-time clock maintained by the SPE decrementer to check buffer progress against a hard, real-time deadline.

When software recognizes the DMA List Command Stall-and-Notify event, it can check the value of the decrementer to find out how much time is left before the next upcoming deadline. If software determines that the remainder of the buffer cannot be transferred before the deadline or that all transfers for the buffer will not finish early enough to allow the SPU to complete some buffer-dependent computation, software can simply cancel the remaining list elements by setting their transfer sizes to ‘0’.

When software cancels unneeded list elements in this way, the system may realize two important benefits. The first benefit is that system power consumption may be reduced by preventing the DMA engine from performing needless work that causes driving and switching internal and possibly external buses. The second benefit is that other time-critical tasks that would have been impeded by competition from the remaining list element transfers may finish earlier because bus bandwidth that would have been wasted on needless transfers becomes available for other, productive transfers.

18.10.3.2 *Linked Data Structure Traversal*

Software can cooperate with the MFC DMA engine to search a complex data structure using a novel application of the DMA List Command Stall-and-Notify event.

Consider the task of searching through some portion of an ordered binary tree, and then performing computation on a matching result. In this example, each node in the tree contains a data element, along with pointers to left and right children. In the tree, internal nodes have a valid pointer for either child, while leaf nodes have left and right child pointers equal to the null pointer. Further, for the purposes of this discussion, we assume that the storage for each tree node has been allocated from a heap in some arbitrary order; for example, the tree was created from randomly ordered data.

A straight-forward implementation coded in a high-level language might fetch a parent node and examine its data to determine if it matches the search key. If a match is found, then the search terminates and the compute phase of the program is started. if a match is not found, either the left or right child pointer is followed depending on whether the search key is logically greater than the current node’s data. The search terminates when a match is found, or when the traversal can proceed no further (the child pointer to be followed is the null pointer). This approach has the main disadvantage of serializing tree traversal, node fetching, and computation.

One alternative is to have the DMA engine and a handler for the stall-and-notify event cooperate to asynchronously search for a match in the tree while the main program independently computes. The steps in this cooperative process are as follows:

1. Software reserves storage for a DMA list with number of list elements equal to the maximum number of nodes to be searched (the maximum tree depth). Note that individual DMA list elements can be modified after the DMA list command is queued, but the number of elements in the list cannot.
2. For the first element in the DMA list, software sets the effective address to that of the parent node and sets the element’s stall-and-notify flag.
3. Software initiates the search operation by enqueuing the DMA list **getl** command and then proceeds to process an independent task; software is assumed to have set up interrupt handling and the DMA List Command Stall-and-Notify event handler at an earlier time.
4. When the interrupt for the stall-and-notify event is taken, a node will have been transferred into SPU LS. The DMA List Command Stall-and-Notify event handler will compare the node’s

data against the search key, and it terminates the remaining elements in the DMA list—by setting their transfer sizes to ‘0’—if the data and key match (search success).

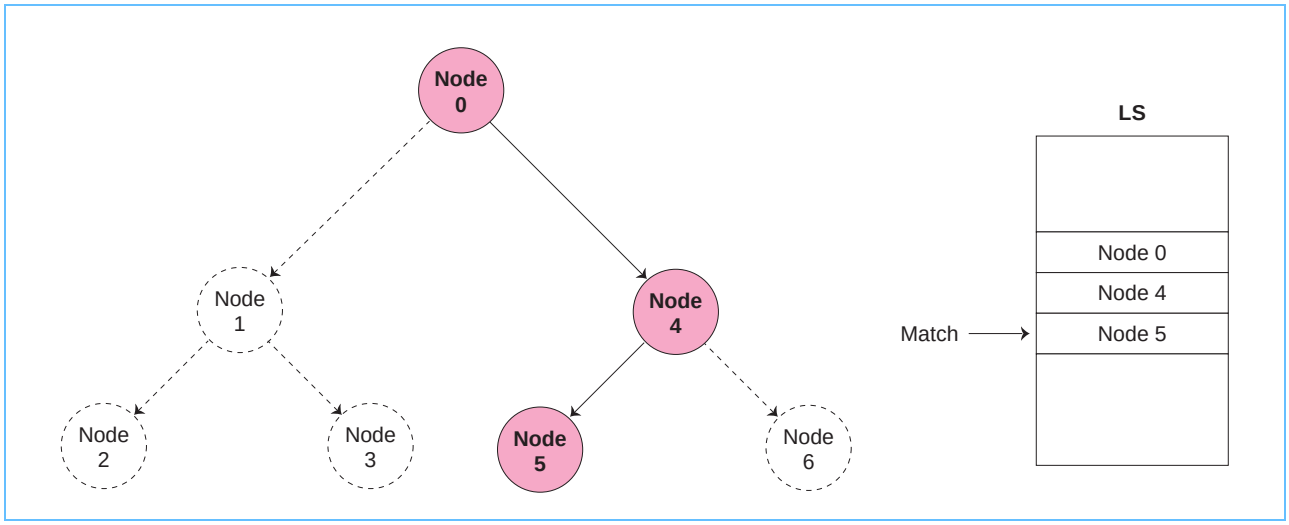
5. If the node’s data is logically greater than the search key and the left child pointer is not the null pointer, then copy the left child pointer to the effective address of the next element in the DMA list, set the element’s stall-and-notify flag, acknowledge the event, and return from the handler to the interrupted program. If the left child pointer is the null pointer, then terminate the remaining elements in the DMA list (search failure).
6. If the node’s data is logically less than the search key and the right child pointer is not the null pointer, then copy the right child pointer to the effective address of the next element in the DMA list, set the element’s stall-and-notify flag, acknowledge the event, and return from the handler to the interrupted program. If the right child pointer is the null pointer, then terminate the remaining elements in the DMA list (search failure).

Some combination of steps 4-7 will be repeated until the search succeeds or fails. There are at least two benefits to this search method. First, SPU efficiency is improved because all the time spent transferring data into SPU LS can be overlapped with useful SPU computation. Second, a recursive algorithm, which may fail due to a limit on the size of the call stack, is avoided.

Figure 18-1 on page 501 shows graphically how a search might proceed. First, the algorithm loads node 0 into LS, fails to find a match, and proceeds to the right child. Next, the algorithm loads the right child, node 4, into LS, fails to find a match, and proceeds to the left child. The algorithm loads the left child, node 5 into LS and finds a match. The nodes are loaded into LS autonomously by the DMA engine; checking for a match and setting up the next transfer is done by the event handler. Software performs a minimum of processing while letting the DMA engine do the part of the algorithm it was designed to do well.

In *Figure 18-1*, the nodes are shown being loaded into separate areas of LS due to the limitations of a static drawing. In a real implementation, a single LS area can be used to hold all nodes because only one node is held in LS at a time.

Figure 18-1. SPE Search of Ordered Binary Tree



SPE software and MFC hardware can use the above algorithm to support multiple simultaneous searches by having the main program initiate several DMA list commands, each with a unique tag group identifier. The DMA List Command Stall-and-Notify handler can maintain a list of active

searches and identify which searches have events pending by reading the DMA List Stall-and-Notify Tag Status channel. The handler can resume or terminate each search according to the above algorithm.

18.10.4 MFC SPU Command Queue Available Event

When software has more than sixteen MFC commands to issue, it can issue the first sixteen but then it must wait until space in the queue becomes available to enqueue the remaining commands. If the SPU has no other work to do, it can wait for space by stalling on a write to the MFC Class ID and Command Opcode channel. If the SPU can proceed with other tasks until space in the MFC queue becomes available, software can set up a handler for the MFC SPU Command Queue Available event so that more MFC commands are enqueued as soon as space is available even though the SPU is attending to other tasks.

Software can set up a queue in LS for MFC commands that are ready for processing. The MFC SPU Command Queue Available event handler takes commands from this queue as MFC queue space becomes available. When the handler finds an empty software queue, it can set a flag that lets the application software know that it must take action to inform the handler that commands are available to be sent to the MFC queue. The software queue can be made large enough to hold the maximum number of commands generated by application software at any one time. Thus, application software can be written under the assumption that the MFC queue is much larger than it actually is. The software queue and the event handler work together to create this virtual MFC queue.

18.10.5 SPU Read Inbound Mailbox Available Event

The SPU Read Inbound Mailbox Available event is raised when a message value is written to the inbound mailbox queue. The message can be any 32-bit value, and messages of more than 32 bits can be sent by enqueuing multiple 32-bit words in succession. Software can define a formal message-passing protocol that requires all messages to begin with header word containing message type and length information. The message decoding logic can be encapsulated in the event handler.

For example, consider an application where effective-address and transfer-size information for a DMA command is sent by another processor or device to an SPU through its mailbox. The SPU Read Inbound Mailbox Available event handler can read the mailbox data, decode the message, determine that it should handle the message, and initiate a DMA transfer; all this processing can proceed with minimal impact on the SPU as it processes an independent task. Having the handler interact with the MFC Tag-Group Status Update event handler and its data structures could increase the efficiency and capability of these asynchronous handlers by allowing tag groups to be used when appropriate.

18.10.6 SPU Signal-Notification Available Event

An SPU Signal-Notification 1 Available or SPU Signal-Notification 2 Available events is raised when a signal value is written to a respective signal-notification register in the MFC's MMIO address space. Each signal value is a 32-bit word much like a mailbox message value. Unlike mailbox queues, signal notification registers can operate in an OR-mode that allows the successive values written to the registers to be accumulated with a logical OR operation.

OR mode allows the single 32-bit signal value to be partitioned so that multiple short messages can be received simultaneously. For example, when software on the SPU and software on another processor participate in a producer-consumer relationship, they need to send and receive simple notices such as a buffer-ready flag. In the simplest case, 32 different flags can be implemented in a single 32-bit signal value.

Since the response times of participants in distributed producer-consumer relationships is usually unpredictable, SPE software can process independent tasks while the SPU Signal Notification Available event handlers respond to signals and perform the simple protocol steps that keep the producer-consumer relationship alive.

18.10.7 Lock-Line Reservation Lost Event

The Lock-Line Reservation Lost event is raised when the PPE or other device writes to any of the addresses in the lock-line reserved by the **getllar** command. A Lock-Line Reservation Lost event handler can be used by an application to receive asynchronous notification that a shared-memory synchronization predicate or condition may have been satisfied.

For example, atomic commands such as the **getllar** command could be executed to reserve a lock-line containing the address of a mutual-exclusion (mutex) lock, also called a semaphore. When the PPE or other device writes to the mutex lock, the lock-line reservation will be lost resulting in the Lock-Line Reservation Lost event being raised. In a micro-thread environment, where the SPU implements software multi-threading, the event handler could consult a data structure to see that the event allows a previously suspended thread that needs to use the mutex lock to be re-awakened. The handler can interact with the thread-management kernel to move the thread from the suspended list to a list that can be run. A version of the POSIX thread library interfaces `pthread_cond_wait(3)` or `pthread_mutex_lock(3)` for the SPU might use this event to implement efficient user-level threading.

18.10.8 Privileged Attention Event

The Privileged Attention event is raised when the PPE or other device sets the Attention Event Required bit is set in the SPU Privileged Control register. This event is intended to signal an urgent request for SPU action. The meaning of this event depends on the operating system environment and the organization of resources in the system. One possible meaning for the event is that the SPU has not responded as expected to the passing of a deadline; in this case, the event could signal to SPE software that it must respond in a certain way to prevent the PPE from terminating the SPU task(s).

19. DMA Transfers and Inter-Processor Communication

19.1 Introduction

Being a multiprocessor system on a chip, the CBE processor has many attributes of a shared-memory system. The PPE and all SPEs have coherent access to main storage. But the CBE processor is not a traditional shared-memory multiprocessor. For example, an SPE can execute programs and directly load and store data only from and to its private local store (LS). In a traditional shared-memory multiprocessor, data communication and synchronization among processors happen at least partially as a side-effect of the fact that all processors use the same shared memory.

Since SPEs lack shared memory, they must communicate explicitly with other entities in the system using three primary communication mechanisms: DMA transfers, mailbox messages, and signal-notification messages. All three communication mechanisms are implemented and controlled by the SPE's MFC. *Table 19-1* summarizes the three primary mechanisms. This chapter describes these mechanisms, plus additional MFC mechanisms for synchronizing and otherwise managing DMA transfers and related MFC functions. The channel interface used by an SPE to initiate and monitor these mechanisms is described in *Section 17 SPE Channel and Related MMIO Interface* on page 441.

Table 19-1. SPE DMA and inter-Processor Communication Mechanisms

Mechanism	Description
DMA transfers	Used to move data and instructions between main storage and an LS. SPEs rely on asynchronous DMA transfers to hide memory latency and transfer overhead by moving information in parallel with SPU computation.
Mailboxes	Used for control communication between an SPE and the PPE or other devices. Mailboxes hold 32-bit messages. Each SPE has two mailboxes for sending messages and one mailbox for receiving messages.
Signal notification	Used for control communication from the PPE or other devices. Signal notification (also called *signaling)* uses 32-bit registers that can be configured for one-sender-to-one-receiver signalling or many-senders-to-one-receiver signalling.

One type of programming model may rely on the PPE to perform the task of application management by assigning and distributing work to the SPEs. A significant part of this task may be loading main storage with programs and data and then notifying an SPE of available work by either writing to its mailbox or one of its signal-notification registers. After getting the message or signal, the SPE performs a DMA operation to transfer the data and code to its LS. In a variation on this programming model, the PPE might perform the DMA operation and then send a message or signal to the SPE when the DMA operation completes.

After processing the data, an SPE can use another DMA operation to deliver the results to main storage. When the DMA transfer of the SPE results from LS to main storage finishes, the SPE can write a completion message to one of its two outgoing mailboxes that informs the PPE that processing and delivery is complete. If the completion message requires more than 32 bits of information, the SPE can write multiple mailbox messages or use a DMA operation to transfer the long message to main storage where the PPE can read it. Even when a long completion message is transferred with a DMA operation, an outgoing mailbox message can be used to inform the PPE that the message is available.

Although signal notifications and mailbox messages are similar, there are important differences. *Table 19-2* summarizes the differences between using a mailbox and using a signal to communicate.

Table 19-2. Comparison of Mailboxes and Signals

Attribute	Mailboxes	Signals
Direction	One inbound, two outbound, all accessible through channel interface.	Two inbound accessible through channel interface, but can send signal using MFC send-signal commands.
Interrupts	One mailbox can interrupt PPE. Two mailbox-available Event interrupts.	Two signal-notification Event interrupts.
Message Accumulation	No.	Yes: over-write mode (one-to-one), logical OR mode (many-to-one).
Unique SPU Commands	No; programs use channel reads and writes.	Yes, **sndsig**, **sndsigf**, and **sndsigb** for sending signals to other units. See *Section 19.2.3* on page 510.
Destructive Read	Reading a mailbox consumes an entry.	Reading a channel resets all 32 bits to ‘0’.
Channel Count	Indicates number of available entries.	Indicates waiting signal.
Number	Three mailboxes: 4-deep incoming, 1-deep outgoing, 1-deep outgoing with interrupt.	Two signal registers.

19.2 MFC Commands

One of the functions of an MFC is to act as a specialized co-processor for its associated SPU. An MFC has the ability to execute operations from its command set, and it executes them autonomously. The MFC maintains two queues to hold issued commands: the 16-entry MFC SPU Command Queue and the 8-entry MFC Proxy Command Queue. MFC commands issued by the SPU are buffered only in the SPU Command Queue; MFC commands issued by the PPE or other system devices are buffered only in the Proxy Command Queue.

An SPU uses its channel interface to send commands to its associated MFC. The SPU can send two types of commands: immediate commands and queueable commands. Queueable commands are entered into the MFC SPU Command Queue and are processed by the MFC at a time determined by two factors: the MFC execution algorithm and any barrier or fence modifiers attached to commands in the queue. When possible and beneficial, the MFC will execute commands out-of-order. Immediate commands are executed when issued by the SPU and are not entered into a queue.

Other devices in the system, including the PPE, use the MFC’s MMIO (memory-mapped-I/O) registers to send commands to a particular SPU’s associated MFC. The MFC accepts only queueable commands through the MMIO registers. Queueable commands are entered into the MFC Proxy Command Queue, and the MFC processes these commands as it does commands in the SPU Command Queue: possibly out of order to improve efficiency but respecting barriers and fences. Commands issued through the MMIO registers to the Proxy Command Queue are called Proxy Commands.

The characteristics of the two command queues are shown in *Table 19-3*. These two queues are independent; the MFC chooses commands to execute from a queue without regard to the activity or conditions in the other queue. Together, these two queues are named the MFC Command Queues (or Queue).

Table 19-3. MFC Command Queues

Queue	Entries	Description
MFC SPU Command Queue	16	For MFC commands sent from the SPU through the channel interface.
MFC Proxy Command Queue	8	For MFC commands sent from the PPE, other SPUs, or other devices through the MMIO registers.

Each MFC has a DMA controller that implements DMA transfers. DMA-transfer commands contain both a Local Store Address (LSA) and an Effective Address (EA) and thereby initiate transfers between the two storage domains. The MFC converts queued DMA-transfer commands into DMA transfers. Each MFC can maintain and process multiple simultaneous MFC commands and multiple simultaneous DMA transfers.

An MFC can also autonomously perform a sequence of DMA transfers in response to a DMA list command issued by software on its associated SPU.

The majority of MFC commands have names that imply a direction, such as **get**, **put**, and **sndsig**.The data-transfer direction for these MFC commands is always referenced from the perspective of the SPU that is associated with the MFC that executes the commands. Thus, **get** commands transfer data into the LS associated with the MFC; **put** commands transfer data out of the LS associated with the MFC.

Software assigns each MFC queueable command with a 5-bit Tag Group ID when the command is issued to the MFC. Software can use Tag Group IDs to monitor the completion of all queued commands in one or more tag groups in a single command queue. SPU software can check the tag-group status by polling, by stalling, or with an asynchronous interrupt.

The MFCs support out-of-order execution of MFC commands. If software needs to maintain order between commands in a queue, it has three ways to do so:

- A command can be issued with its tag-specific barrier feature-bit set to order the command against all preceding and all succeeding commands in the tag group.
- A command can be issued with its tag-specific fence feature-bit set to order the command against all preceding commands in the tag group.
- A separate **barrier** command (*Section 19.2.3* on page 510) can be issued to order the command against all preceding and all succeeding commands in the queue, regardless of tag group.

MFC commands can be classified into two types: DMA commands (*Section 19.2.1*), and synchronization commands (*Section 19.2.3* on page 510). All the DMA commands are queueable; three of the synchronization commands are immediate. The DMA commands can be modified with the suffixes described in *Section 19.2.4* on page 510.

19.2.1 DMA Commands

The majority of MFC commands initiate DMA transfers; these are called DMA commands. The basic **get** and **put** DMA commands and all the possible variants are listed in *Table 19-4* and *Table 19-5*. The command-modifier suffixes for the MFC commands are listed in *Table 19-7*; these suffixes are used to create the variants shown in *Table 19-4* and *Table 19-5*. Since the LSs of the SPEs and the I/O subsystems are typically mapped into the effective address space, DMA commands can transfer data between an LS and these areas as well.

Regardless of the initiator (SPU, PPE, or other device), DMA transfers move up to 16 KB of data between an LS of and main storage. An MFC supports naturally aligned DMA transfer sizes of 1, 2, 4, 8, and 16 bytes and multiples of 16 bytes.

The performance of a DMA transfer can be improved when the source and destination addresses have the same quadword offsets within a 128-byte cache line. Quadword-offset-aligned transfers generate full cache-line bus requests for every cache line, except possibly the first and last. Transfers that start or end in the middle of a cache line transfer a partial cache line in the first or last bus request, respectively.

The performance of a DMA data transfer when the source and destination addresses have different quadword offsets within a cache line is about half that of quadword-aligned transfers, because every bus request is a partial cache-line transfer; in effect, there are two bus requests for each cache line of data.

Peak performance is achieved for transfers in which both the EA and the LSA are 128-byte aligned and the size of the transfer is a multiple of 128 bytes.

Fifteen types of **put** commands move data from LS to main storage. Nine types of **get** commands move data into LS from main storage. For single-transfer DMA code example, see *Section 19.4.1* on page 521.

Table 19-4. MFC DMA Put Commands (Page 1 of 2)

Mnemonic	Possible Initiator		Description
	SPU	PPE	
put	•	•	Moves data from LS to the effective address.
puts		•	Moves data from LS to the effective address and starts the SPU after the DMA operation completes.
putf	•	•	Moves data from LS to the effective address with fence (this command is locally ordered with respect to all previously issued commands within the same tag group and command queue).
putb	•	•	Moves data from LS to the effective address with barrier (this command and all subsequent commands with the same tag ID as this command are locally ordered with respect to all previously issued commands within the same tag group and command queue).
putfs		•	Moves data from LS to the effective address with fence (this command is locally ordered with respect to all previously issued commands within the same tag group and command queue) and starts the SPU after the DMA operation completes.
putbs		•	Moves data from LS to the effective address with barrier (this command and all subsequent commands with the same tag ID as this command are locally ordered with respect to all previously issued commands within the same tag group and command queue) and starts the SPU after the DMA operation completes.

Table 19-4. MFC DMA Put Commands (Page 2 of 2)

Mnemonic	Possible Initiator		Description
	SPU	PPE	
putl	•		Moves data from LS to the effective address using an MFC list.
putlf	•		Moves data from LS to the effective address using an MFC list with fence (this command is locally ordered with respect to all previously issued commands within the same tag group and command queue).
putlb	•	•	Moves data from LS to the effective address using an MFC list with barrier (this command and all subsequent commands with the same tag ID as this command are locally ordered with respect to all previously issued commands within the same tag group and command queue).

Table 19-5. MFC DMA Get Commands

Mnemonic	Possible Initiator		Description
	SPU	PPE	
get	•	•	Moves data from the effective address to LS.
gets		•	Moves data from the effective address to LS, and starts the SPU after the DMA operation completes.
getf	•	•	Moves data from the effective address to LS with fence (this command is locally ordered with respect to all previously issued commands within the same tag group and command queue).
getb	•	•	Moves data from the effective address to LS with barrier (this command and all subsequent commands with the same tag ID as this command are locally ordered with respect to all previously issued commands within the same tag group and command queue).
getfs		•	Moves data from the effective address to LS with fence (this command is locally ordered with respect to all previously issued commands within the same tag group), and starts the SPU after the DMA operation completes.
getbs		•	Moves data from the effective address to LS with barrier (this command and all subsequent commands with the same tag ID as this command are locally ordered with respect to all previously issued commands within the same tag group and command queue), and starts the SPU after the DMA operation completes.
getl	•		Moves data from the effective address to LS using an MFC list.
getlf	•		Moves data from the effective address to LS using an MFC list with fence (this command is locally ordered with respect to all previously issued commands within the same tag group and command queue).
getlb	•		Moves data from the effective address to LS using an MFC list with barrier (this command and all subsequent commands with the same tag ID as this command are locally ordered with respect to all previously issued commands within the same tag group and command queue).

19.2.2 DMA List Commands

A DMA list is sequence of eight-byte list elements, stored in an SPE's LS, each of which describes a DMA transfer. DMA list commands can be initiated only by SPU programs, not by other devices or by programs running on the PPE. DMA lists are accessed when the MFC executes one of the DMA list commands (**getl**, **getlf**, **getlb**, **putl**, **putlf**, or **putlb**) described in *Table 19-4* on page 508 and *Table 19-5* on page 509.

A DMA list command can specify up to 2,048 DMA transfers, each up to 16 KB in length. Thus, a DMA list command can transfer up to 32 MB, which is 128 times the size of the 256-KB LS, more than enough to accommodate future increases in the size of LS. The space required for the maximum-size DMA list is 16 KB (eight bytes for each of 2,048 list elements).

DMA list commands are used to move data between a contiguous area in an SPE's LS and possibly non-contiguous area in the effective address space, thus implementing scatter-gather functions between main storage and the LS.

For details and a DMA list code example, see *Section 19.4.4* on page 527.

19.2.3 Synchronization Commands

MFC synchronization commands, listed in *Table 19-6*, are used to control the order in which storage accesses are performed. The synchronization commands include four atomic commands (**getllar**, **putllc**, **putlluc**, and **putqlluc**), three send-signal commands (**sndsig**, **sndsigf**, and **sndsigb**), and three barrier commands (**mfcsync**, **mfceieio**, and **barrier**). The MFC can issue up to two outstanding atomic commands, one immediate-type and one **putqlluc**. For details on the four atomic commands, see *Section 20.3 SPE Atomic Synchronization* on page 587.

Table 19-6. MFC Synchronization Commands

Command	Possible Initiator		Executed Immediately	Description
	SPU	PPE		
getllar		•	•	Get lock line and reserve.
putllc		•	•	Put lock line conditional.
putlluc		•	•	Put lock line unconditional.
putqlluc	•	•		Put queued lock line unconditional.
barrier	•	•		Barrier type ordering. Ensures ordering of all preceding DMA commands with respect to all commands following the barrier command in the same command queue. The **barrier** command has no effect on the immediate DMA commands: **getllar**, **putllc**, and **putlluc**.
mfceieio	•	•		Controls the ordering of **get** commands with respect to **put** commands, and of **get** commands with respect to **get** commands accessing storage that is caching inhibited and guarded. Also controls the ordering of **put** commands with respect to **put** commands accessing storage that is memory coherence required and not caching inhibited.
mfcsync	•	•		Controls the ordering of DMA **put** and **get** operations within the specified tag group with respect to other processing units and mechanisms in the system.
sndsig	•	•		Write Signal Notification Register in another device.
sndsigf	•	•		Write Signal Notification Register in another device, with fence.
sndsigb	•	•		Write Signal Notification Register in another device, with barrier.

19.2.4 Command Modifiers

The suffixes listed in *Table 19-7* are included in the mnemonics of DMA commands listed in *Table 19-4* and *Table 19-5*. These suffixes extend or refine the function of a command. For example, a **put** command moves data from LS to the effective address. A **puts** command moves data from LS to an effective address and starts the SPU after the DMA transfer completes.

Commands with an **s** suffix can only be issued to the MFC Proxy Command Queue. Commands with an **l** suffix and all the MFC atomic commands can only be issued to the MFC SPU Command Queue. Commands with an **f** or **b** suffix can be issued to either command queue.

Table 19-7. MFC Command-Modifier Suffixes

Mnemonic	Possible Initiator		Description
	SPU	PPE	
s		•	**Start SPU.** Starts the SPU running at the address in the SPU Next Program Counter Register (SPU_NPC) after the MFC command completes.
f	•	•	**Tag-specific fence.** Command is locally ordered with respect to all previously issued commands in the same tag group and command queue.
b	•	•	**Tag-specific barrier.** Command is locally ordered with respect to all previously issued and all subsequently issued commands in the same tag group and command queue.
l	•		**List command.** Command processes a list of DMA list elements located in LS. Up to 2048 elements in a list; each list element specifies a transfer of up to 16 KB.

19.2.5 Tag Groups

All DMA commands except **getllar**, **putllc**, and **putlluc** can be tagged with a 5-bit Tag Group ID. Tag Group IDs on commands in the SPU Command Queue are independent of the Tag Group IDs on commands in the Proxy Command Queue. Software can ignore the Tag Group ID capabilities or it can tag one or more commands with the same Tag Group ID to form a tag group. When multiple commands are in the same tag group, SPU software can use the Tag Group ID to take advantage of several tag-related features including: checking on the status of a tag group, enabling an interrupt that is raised when one or more tag groups complete, and enforcing an execution order of commands in a tag group.

Programmers can enforce ordering among DMA commands in a tag group with a fence or barrier option by appending an **f** or **b**, respectively, to the command mnemonic (see *Section 19.2.4* and *Table 19-7*). A fenced command is not executed until all previously issued commands within the same tag group have been performed; commands issued after the fenced command may be executed before the fenced command. A barrier command and all the commands issued after the barriered command are not executed until all previously issued commands in the same tag group have been performed.

19.2.5.1 *Status of Tag Groups Containing MFC Commands*

The SPE uses the following channels, listed in *Table 17-2* on page 443, to query tag groups in the MFC SPU Command Queue:

- `MFC_WrTagMask`
- `MFC_WrTagUpdate`
- `MFC_RdTagStat`
- `MFC_RdListStallStat`
- `MFC_WrListStallAck`

The PPE uses the following MMIO registers, listed in *Table 17-2* on page 443, to query tag groups in the MFC Proxy Command Queue:

- `Prxy_QueryType`
- `Prxy_QueryMask`
- `Prxy_TagStatus`

The MFC SPU Command Queue has tag groups 0 to 31, and these are different from MFC Proxy Command Queue tag groups 0 to 31. When one or more commands belonging to a tag group are queued in the MFC, the status of that tag group is not empty. When all commands belonging to a tag group are processed and completed, the status of that tag group becomes empty.

Software can use tag-group identifiers to check the completion status of all queued commands in one or more tag groups. For details and examples, see *Section 19.3.3* on page 517 and *Section 19.4.3* on page 523.

19.2.5.2 *MFC Commands with Tag-Group Dependencies*

The MFC command-execution mechanism checks tag group dependencies in the MFC SPU Command Queue separately from tag-group dependencies in the MFC Proxy Command Queue; a tag group can have members from only one queue. Tag-group dependencies are determined by the opcode of the command, the command's tag ID, and conditions in the appropriate command queue. Dependencies on still-active commands are cleared as each command is completed.

The **get**, **put**, and **sndsig** commands with **f** (fence) or **b** (barrier) suffixes create a tag-group-specific dependency for the command as compared with other commands in the same tag group and the same command queue.

The **putqlluc** command has a tag-group-specific dependency as compared with other commands in the MFC SPU Command Queue and a non-tag-specific dependency as compared with other **putqlluc** commands in the MFC SPU Command Queue.

Note: *The Cell Broadband Engine Architecture specifies that the **putqlluc** command has an implied tag-specific fence which prevents this command from being issued until all previously issued commands with the same tag have completed. The CBE processor implementation further adds a fence against all other **putqlluc** commands so that all **putqlluc** commands are ordered regardless of their tag IDs.*

Note: *The Cell Broadband Engine Architecture specifies that the **mfceieio** and **mfcsync** commands are tag-specific. This means that these commands and all subsequent commands in the same tag group are ordered with respect to all previous commands in the queue with the same tag ID. However, the CBE processor implementation treats these commands are non-tag-specific such that they are ordered with respect to all subsequent and previous commands in the command queue regardless of their tag group.*

19.2.6 MFC Command Issue

The MFC supports out-of-order execution of independent MFC commands. The MFC considers commands that have no dependencies to be eligible for issue. The issue mechanism has two slots: Slot 0 and Slot 1. Each command is assigned to either Slot 0 or Slot 1 depending on the command type. *Table 19-8* lists the MFC command and their slot assignment.

Table 19-8. MFC Issue Slot Command Assignments

Slot	Instructions
Slot 0	put, sndsig, putqlluc, sdcrz, sdcrst, sdcrf
Slot 1	get, sdcrt, sdcrtst, barrier, mfcsync, mfceieio

The MFC issue mechanism alternates issuing commands in Slot 0 and Slot 1 at the EIB frequency, which is half the SPU frequency. When multiple commands in a slot are eligible, the oldest command in the MFC SPU Command Queue has priority; if no commands in the MFC SPU Command Queue are eligible, then the oldest eligible command in the MFC Proxy Command Queue is selected. Thus, commands in the SPU Command Queue have higher priority than commands in the Proxy Queue. Sixteen requests can be outstanding.

19.2.7 Replacement Class ID and Transfer Class ID

The MFC Class ID and Command Opcode channel, listed in the MFC Command Parameters group of *Table 17-2* on page 443, can be used to specify both the Replacement Class ID (`RclassID`) and the Transfer Class ID (`TclassID`) for each MFC command.

The `TclassID` allows application software to influence the allocation of bus bandwidth for an MFC command. The `TclassID` generates an index into the issue-quota table (IQ0, IQ1, and IQ2 fields in the `MFC_TClassID` register, see *Section 17.9.6* on page 455), which specifies the fraction of bus issue slots to allocate to a transfer class.

The `RClassID` allows privileged software to influence L2-cache and TLB replacement for cache misses caused by the MFC command. The `RClassID` is used to generate an index into the Replacement Management Table (RMT), which specifies which members in a congruence class are eligible for replacement. See *Section 6.3 Replacement Management Tables* on page 154 for details.

The MFC Class ID register performs the same function for commands issued to the MFC by the PPE or the SPU. This register is used to influence cache replacement and bus bandwidth allocation associated with an SPU's MFC, and it has no effect on resources associated with the PPE.

The default class ID ('0') is used for all undefined or invalid class IDs. An invalid class ID does not generate an exception.

19.2.7.1 *TClassID Enabled*

By default, the Transfer Class ID facility is disabled and the issue mechanism does not use the `TclassID` value of an MFC command. However, performance of the MFC can be improved if privileged software enables `TclassID` in the privileged `MFC_TClassID` register. Enabling the facility allows software to manage the bus and bandwidth allocation for requests to different types of targets, such as on-chip storage, off-chip storage, and I/O.

When `TclassID` is enabled, the issue mechanism enables round-robin selection between `TClassID0`, `TClassID1`, and `TClassID2` eligible MFC commands for both Slot 0 and Slot 1. Slot 0 and Slot 1 have separate round-robin tokens that track the class of the last command issued in that slot. When Slot Alternation (see below) is disabled for a class, that class is skipped by the Slot 1 round-robin mechanism, and all entries are issued by the Slot 0 round-robin mechanism.

Each TclassID has a quota, specified in the IQ0, IQ1, and IQ2 fields of the MFC_TClassID register, for the number of outstanding requests to the SBI. The sum of these quotas should not exceed sixteen (the depth of the SBI Queue). The quota for each class is used to prevent the issue mechanism from selecting an MFC command in that class when the quota is reached. When the number of outstanding requests for a class is below the quota for that class, the issue mechanism can select eligible commands from the class.

Slot alternation of issues between Slot 0 and Slot 1 can be enabled separately for each TclassID using the SA0, SA1, and SA2 fields of the MFC_TClassID register. Enabling Slot Alternation for a class allows commands from that class to issue every cycle but alternating between Slot 0 and Slot 1 at the EIB frequency. Disabling Slot Alternation for a class forces commands from that class to issue only in Slot 0 every 4th cycle at the EIB frequency.

The suggested uses for Transfer Class IDs are:

- Transfer Class ID 0—For MFC commands that bypass the token request, such as LS to LS (EA translated to LS address) transfers or on-chip MMIO transfers with Slot Alternation enabled (SA0 = 0) to allow **put** and **get** commands to execute at the same time.
- Transfer Class ID 1—For off-chip memory-access MFC commands with Slot Alternation disabled (SA1 = 1) to reduce the penalties associated with switching the direction of the bidirectional memory interface.
- Transfer Class ID 2—For I/O-access MFC commands with Slot Alternation enabled (SA2 = 0) to allow **put** and **get** commands to execute at the same time.

19.2.7.2 *TClassID Disabled*

By default, the Transfer Class ID facility is disabled, and the issue mechanism does not consider the Transfer Class ID when it makes issue decisions. When the Transfer Class ID facility is disabled, the IQ0 (issue quota 0) and SA0 (slot alternation 0) fields in MFC_TClassID affect all MFC commands. IQ0 is sixteen by default to match the depth of the SBI Queue. If software lowers the value of IQ0, the maximum number of outstanding DMA requests to the SBI is reduced accordingly. SA0 is 0 by default to allow the issue mechanism to alternate issuing commands in Slot 0 and Slot 1. If software sets SA0 to 1, this alternation of slots is disabled, and all MFC commands are issued only in Slot 0 every fourth cycle at the EIB frequency.

19.2.8 DMA-Command Completion

When an MFC command from the SPU is complete, the channel count for MFC_Cmd is incremented to indicate that an entry in the MFC SPU Command Queue has become available. When an MFC command from the PPE is complete, the MFC_QStatus register (*Table 19-9* on page 515) is updated. Command completion also updates Prxy_TagStatus register (*Table 17-2* on page 443) if the command's Tag Group is empty.

19.3 PPE-Initiated DMA Transfers

The PPE or other devices can initiate DMA transfers between main storage and an SPE's LS by accessing the MMIO command-parameter registers maintained by the target SPE's MFC. These MMIO registers are listed in *Table 19-9* on page 515.

Table 19-9. MFC Command-Parameter Registers for PPE-Initiated DMA Transfers

Offset From Base	MMIO Register	Max. Entries	R/W	Width (bits)	Functions
x'03004'	MFC_LSA	1	W	32	Specifies the LS address of the DMA transfer.
x'03008'	MFC_EAH	1	W	32	Specifies the high-order half of the EA address of the DMA transfer.
x'0300C'	MFC_EAL	1	W	32	Specifies the low-order half of the EA address of the DMA transfer.
x'03010'	MFC_Size	1	W	16 H	Specifies the size of the DMA transfer.
	MFC_Tag	1	W	16 L	Specifies an identifier for the DMA-transfer command.
x'03014'	MFC_ClassID_CMD	8	W	32	Specifies the opcode, replacement class ID (RclassID), and transfer class ID (TclassID) for the DMA-transfer command.
	MFC_CMDStatus	1	R	32	Returns information about the success or failure of queuing a DMA command.
	MFC_QStatus	1	R	32	Returns the number of queue entries available.

19.3.1 MFC Command Issue

The PPE issues MFC commands by writing parameters to these registers and writing a command opcode to the `MFC_ClassID_CMD` register. To enqueue an MFC command through the MFC Command Parameter Registers, the registers must be written and read in the following sequence:

1. Write the `MFC_LSA` Register. The `MFC_LSA` register must be written with a single 32-bit store.
2. Write the `MFC_EAH` and `MFC_EAL` registers. The `MFC_EAH` register defaults to '0' and need not be written for effective addresses less than 4 GB. Software can write `MFC_EAH` and `MFC_EAL` as two 32-bit stores or one 64-bit store.
3. Write the `MFC_Size` and `MFC_Tag` Parameters. The `MFC_Size` parameter occupies the upper 16 bits of a 32-bit word; `MFC_Tag` occupies the lower 16 bits. `MFC_Size` and `MFC_Tag` can be written with a single 32-bit store or they can be written along with `MFC_ClassID_CMD` in a single 64-bit store.
4. Write the `MFC_ClassID` and `MFC_CMD` Parameters. The `MFC_ClassID` parameter occupies the upper 16 bits of the `MFC_ClassID_CMD` word; MFC_CMD parameter occupies the lower 16 bits of a 32-bit word. `MFC_ClassID` and `MFC_CMD` can be written with a single 32-bit store or they can be written along with the `MFC_Size` and `MFC_Tag` parameters in a single 64-bit store.
5. Read the `MFC_CMDStatus` Register. The read of the `MFC_CMDStatus` register is a 32-bit load.
6. If there was insufficient space in the queue, wait until there is space and retry the sequence starting at step 1.
7. If a sequence error occurred, retry sequence starting at step1.

The least-significant two bits of the command-status value returned from the read of the `MFC_CMDStatus` register indicates success or error when attempting to enqueue an MFC command, as shown in *Table 19-10*.

Table 19-10. MFC_CMDStatus Return Values

Least-Significant Two Bits	Command Status
0	The Enqueue was successful.
1	A sequence error occurred while enqueuing the DMA transfer (for example an interrupt occurred, then another DMA transfer was started within an interrupt handler). In this case, the MFC-command enqueue must be restarted from the beginning of the command issue protocol.
2	The enqueue failed due to insufficient space in the command queue.
3	Both of the above errors occurred.

Figure 17-2 on page 452 shows a flowchart of the sequence in which MFC commands are issued by PPE software.

19.3.1.1 *Example: A PPE-Initiated DMA Transfer*

The following C-language program fragment shows how code running on the PPE can initiate a 16-KB DMA transfer from EA 0x'10000000' to LS address 0x'500' using a tag ID of 5.

```
void *ps = get_ps();
unsigned int ls = 0x500;
unsigned int long long ea = 0x10000000;
unsigned int size = 0x4000;
unsigned int tag = 5;
unsigned int classid = 0;
unsigned int cmd = MFC_GET_CMD;
unsigned int cmd_status;

do {
    *((volatile unsigned int *)(ps + MFC_LSA)) = ls;
    *((volatile unsigned long long *)(ps + MFC_EAH)) = ea;
    *((volatile unsigned int *)(ps + MFC_Size)) = (size << 16) | tag;
    *((volatile unsigned int *)(ps + MFC_ClassID)) = (classid << 16) | cmd;
    /*
     * Read MFC_CMDStatus to enqueue command and check enqueue success.
     */
    cmd_status = *((volatile unsigned int *)(ps + MFC_CMDStatus)) & 0x3;

} while (cmd_status); /* Attempt to enqueue until success */
```

19.3.2 MFC Command-Queue Control Registers

The PPE can control and monitor an SPE by using the MFC command-queue control registers, which are listed in *Table 19-11* on page 517. These MMIO registers are in the SPE problem-state memory region of each SPE along with the MFC Command Parameter Registers.

Table 19-11. MFC Command-Queue MMIO Registers for PPE-Initiated Commands

Offset From Base	MMIO Register	Max. Entries	R/W	Width (bits)	Functions
x'03014'	MFC_CMDStatus	1	R	32	Returns information about the success or failure of queuing a DMA command.
	MFC_QStatus	1	R	32	Returns the number of queue entries available.
x'0321C'	Prxy_QueryMask	1	R/W	32	Specifies the tag groups to be included in a query operation. Returns the current mask value.
x'03204'	Prxy_QueryType	1	R/W	32	Specifies a tag-group query-completion condition. Returns query status. A return value of '0' means query request complete.
x'0322C'	Prxy_TagStatus	1	R	32	Returns the status of the tag groups enabled in the Prxy_QueryMask register.

19.3.3 DMA-Command Issue Status and Errors

The progress of PPE-initiated DMA commands can be monitored in various ways including checking the completion status of tag groups and checking the number of entries available in the MFC Proxy Command Queue. PPE-initiated DMA commands can also fail due to exceptional conditions that include address-translation faults, command errors, and alignment errors. These exceptional conditions are detected and handled by privileged software executing on the PPE.

19.3.3.1 *DMA Tag-Group Completion*

Each of the 32 tag groups (*Section 19.2.5* on page 511) is assigned a bit in the 32-bit `Prxy_QueryMask` register. When software needs to check the completion status of one or more tag groups, it sets the `Prxy_QueryMask` register with a value that has the corresponding bit set for each DMA tag group of interest. Tag group 31 is assigned the most-significant bit of `Prxy_QueryMask`, and tag group 0 is assigned the least-significant bit.

Software can use three basic protocols to determine the completion status of one or more tag groups:

- Poll Proxy Tag-Group Status Register.
- Poll Proxy Tag-Group Query-Type Register.
- Wait for Tag Group Completion Interrupt.

These protocols are discussed in the following sections.

Poll Proxy Tag-Group Status Register

The basic procedure to poll for the completion of an MFC command or group of MFC commands using the Proxy Tag-Group Status Register is as follows:

1. Issue the MFC commands to the MFC Proxy Command Queue.
2. Set the Proxy Tag-Group Query-Mask Register to the group(s) of interest.

3. Issue an **eieio** instruction prior to reading the Proxy Tag-Group Status Register to ensure the effects of all previous stores complete.
4. Read the Proxy Tag-Group Status Register.
5. If the value is non-zero, at least one of the tag groups of interest has completed. If polling for all the tag groups of interest to complete, XOR the tag group status value with the tag group query mask. A result of ‘0’ indicates that all groups of interest are complete.
6. Repeat steps 4 and 5 until the tag groups of interest are complete.

The following C-language program fragment shows how to poll for tag group completion of a single tag group. This example assumes that MFC commands have already been issued with a tag-group identifier of 5.

```
void *ps = get_ps();
unsigned int tag_mask = 1 << 5;
unsigned int tag_status;

*((volatile unsigned int *)(ps + Prxy_QueryMask)) = tag_mask;

__asm__(“eieio”); /* force write to Prxy_QueryMask to complete */

do {
    tag_status = *((volatile unsigned int *)(ps + Prxy_TagStatus));
} while (!tag_status);
```

The following C-language program fragment shows how to poll for tag group completion of any tag group among many. This example assumes that MFC commands have already been issued with a tag ID of 5, 14, and 31.

```
void *ps = get_ps();
unsigned int tag_mask = (1<<5)|(1<<14)|(1<<31);
unsigned int tag_status;

*((volatile unsigned int *)(ps + Prxy_QueryMask)) = tag_mask;

__asm__(“eieio”); /* force Prxy_QueryMask write to complete */

do {
    tag_status = *((volatile unsigned int *)(ps + Prxy_TagStatus));
} while (!tag_status);
```

The following C-language program fragment shows how to poll for tag group completion of all tag groups among many. This example assumes that MFC commands have already been issued with a tag id of 5, 14, and 31.

```
void *ps = get_ps();
unsigned int tag_mask = (1<<5)|(1<<14)|(1<<31);
unsigned int tag_status;
```

```
*((volatile unsigned int *)(ps + Prxy_QueryMask)) = tag_mask;

__asm__("eieio");

do {
    tag_status = *((volatile unsigned int *)(ps + Prxy_TagStatus));
} while (tag_status ^ tag_mask);
```

Poll Proxy Tag-Group Query-Type Register

The basic procedure to poll for the completion of an MFC command or group of MFC commands using the Proxy Tag-Group Query-Type Register is as follows:

1. Issue the MFC commands to the MFC Proxy Command Queue.
2. Set the Proxy Tag-Group Query-Mask Register to the group(s) of interest.
3. Request a tag group query by writing a value to the Proxy Tag-Group Query-Type Register: write b'01' to indicate a query for completion of any enabled tag group, or write b'10' to indicated a query for completion of all enabled tag groups.
4. Read the Proxy Tag-Group Query-Type Register.
5. If the value is '0', then the requested tag group query condition has been met. If the value is non-zero, then repeat step 4 until it returns a value of '0'.

The following C-language program fragment shows how to poll for tag group completion of any tag group. The code will poll for completion of any of the groups enabled by the tag mask; to poll for completion of all of the groups enabled by the tag mask, a tag group query value of b'10' is set. This example assumes that MFC commands have already been issued with a tag id of 5, 14, and 31.

```
void *ps = get_ps();
unsigned int tag_mask = (1<<5)|(1<<14)|(1<<31);
unsigned int query_type = 1; /* Use 2 for "all" query. */
unsigned int query_status;

*((volatile unsigned int *)(ps + Prxy_QueryMask)) = tag_mask;
*((volatile unsigned int *)(ps + Prxy_QueryType)) = query_type;

__asm__("eieio");

do {
    query_status = *((volatile unsigned int *)(ps + Prxy_QueryType));
} while (query_status);
```

Note: *Waiting for any tag-group completion with a 0 tag mask can result in a hang or deadlock condition.*

Wait for Tag Group Completion Interrupt

The basic procedure to wait for a Tag Group Completion Interrupt is as follows:

1. Enable the Tag Group Completion Interrupt by means of PPE privileged software.
2. Issue the MFC commands to the MFC Proxy Command Queue.
3. Set the Proxy Tag-Group Query-Mask Register to the group(s) of interest.
4. Request a tag group query by writing a value to the Proxy Tag-Group Query-Type Register: write b‘01’ to indicate a query for completion of any enabled tag group, or write b‘10’ to indicated a query for completion of all enabled tag groups.
5. Wait for the Tag Group Completion Interrupt to occur.
6. PPE privileged software must reset the interrupt status after the interrupt is taken.

The following C-language program fragment shows how to setup up for a Tag Group Completion Interrupt on completion of any tag group. As with the previous example, the tag group query value can be changed to b‘10’ to interrupt on completion of all tag groups. This example assumes that PPE privileged software has already enabled the Tag Group Completion Interrupt, the application has already enabled an interrupt handler, and that MFC commands have already been issued with a tag id of 5, 14, and 31.

```
void *ps = get_ps();
unsigned int tag_mask = (1<<5)|(1<<14)|(1<<31);
unsigned int query_type = 1; /* Use 2 for all query. */

*((volatile unsigned int *)(ps + Prxy_QueryMask)) = tag_mask;
*((volatile unsigned int *)(ps + Prxy_QueryType)) = query_type;
```

Because interrupts are asynchronous, the application can proceed with other tasks while the MFC hardware monitors the tag-group completion status.

19.3.3.2 ***MFC Proxy Command Queue Space***

Each MFC has an 8-entry MFC Proxy Command Queue for commands initiated by the PPE. As the PPE submits MFC commands, the queue may become full. When the queue is full, PPE software must wait for space to become available before subsequent commands can be issued.

The MFC Queue Status register contains the current status of the MFC Proxy Command Queue. The most-significant bit is set when the queue is empty. The least-significant 16 bits indicate the number of available slots in the queue. A value of ‘0’ in this field indicates that the queue if full. Software can use the value from this field to set a loop count for the number of MFC commands to enqueue.

The following C-language program fragment shows how to either wait for the MFC Proxy Command Queue to be empty or wait for a specific number of entries, specified in the `space` variable, to become available.

```
void *ps = get_ps();
unsigned int queue_status;
```

```
unsigned int space;
  /*
  * When the requested space is greater than eight
  * wait for MFC Proxy Command Queue to become empty.
  */
if (space > 8)
{
    do {
        queue_status = *((volatile unsigned int *)(ps + MFC_QStatus));
    } while (!(queue_status & 0x80000000));
}
else
{
    do {
        queue_status = *((volatile unsigned int *)(ps + MFC_QStatus));
    } while ((queue_status & 0xFFFF) < space);
}
```

19.4 SPE-Initiated DMA Transfers

The SPE can initiate DMA transfers between main storage and its LS by accessing its channels, listed in *Table 19-12*.

Table 19-12. MFC Command-Parameter Channels for SPE-Initiated DMA Transfers

SPE Channel #	Mnemonic	Max. Entries	Blocking	R/W	Width (bits)	Functions
16	MFC_LSA	1	no	W	32	Specifies the LS address of the DMA transfer.
17	MFC_EAH	1	no	W	32	Specifies the high-order half of the EA address of the DMA transfer.
18	MFC_EAL	1	no	W	32	Specifies the low-order half of the EA address of the DMA transfer.
19	MFC_Size	1	no	W	16	Specifies the size of the DMA transfer.
20	MFC_TagID	1	no	W	16	Specifies an identifier for the DMA-transfer command.
21	MFC_Cmd	16	yes	W	32	Specifies the opcode, replacement class ID (RclassID), and transfer class ID (TclassID) for the DMA-transfer command.

19.4.1 MFC Command Issue

The SPE issues MFC commands by writing parameters to these channels and then writing a command opcode to the `MFC_Cmd` channel, using the **wrch** instruction, described in *Section 17.1.5* on page 446. To enqueue a DMA command, the first five parameters listed in the sequence below can be written in any order, but they must all be written before the `MFC_Cmd` parameter, which must be written last. The one exception is the `MFC_EAH` parameter, which can be skipped if the default value of ‘0’ is acceptable.

1. Write `MFC_LSA` channel with the LS address.
2. Write `MFC_EAH` channel with the high 32 bits of the effective address (defaults to ‘0’ if not written).
3. Write `MFC_EAL` channel with the low 32-bits of the effective address.
4. Write `MFC_Size` channel with the transfer size.
5. Write `MFC_TagID` channel with the tag-group identifier value.
6. Write `MFC_Cmd` channel with the transfer- and replacement-class IDs and the command opcode.

Figure 17-2 on page 452 shows a flowchart of the sequence in which MFC commands are issued by SPE software.

19.4.1.1 *Example: Initiating a DMA Transfer from the SPE*

The following examples shows how to initiate a DMA transfer from an SPE.

```
extern void dma_transfer(volatile void *lsa,     // local store address
                         unsigned int eah,       // high 32-bit effective address
                         unsigned int eal,       // low 32-bit effective address
                         unsigned int size,      // transfer size in bytes
                         unsigned int tag_id,    // tag identifier (0-31)
                         unsigned int cmd);      // DMA command
```

An ABI-compliant assembly-language implementation of the subroutine is:

```
        .text
        .global      dma_transfer
dma_transfer:
        wrch         $MFC_LSA, $3
        wrch         $MFC_EAH, $4
        wrch         $MFC_EAL, $5
        wrch         $MFC_Size, $6
        wrch         $MFC_TagID, $7
        wrch         $MFC_Cmd, $8
        bi           $0
```

A comparable C implementation using the SPU composite intrinsic, **spu_mfcdma64**, is:

```
#include <spu_intrinsics.h>

void dma_transfer(volatile void *lsa, unsigned int eah, unsigned int eal,
                         unsigned int size, unsigned int tag_id, unsigned int cmd)
{
        spu_mfcdma64(lsa, eah, eal, size, tag_id, cmd);
}
```

For an example of enqueuing a set of DMA transfers using a DMA list command, see *Section 19.4.4* on page 527.

19.4.2 MFC Command-Queue Monitoring Channels

SPE software can monitor the MFC commands in one or more tag groups with the MFC Tag Status channels, listed in *Table 19-13*. An SPE can monitor external events with the MFC Event channels, listed in *Table 19-14*. The following examples show how to use these channels.

Table 19-13. MFC Tag-Status Channels for Monitoring SPE-Initiated Commands

SPE Channel #	Mnemonic	Max. Entries	Blocking	R/W	Width (bits)	Functions
22	MFC_WrTagMask	1	no	W	32	Specifies the tag groups to be included in a query operation.
23	MFC_WrTagUpdate	1	yes	W	32	Specifies when the MFC tag-group status is updated in MFC_RdTagStat.
24	MFC_RdTagStat	1	yes	R	32	Returns the status of the tag groups from the last tag-group status update request.
25	MFC_RdListStallStat	1	yes	R	32	Returns the tag groups that have an MFC DMA list command in the stall state.
26	MFC_WrListStallAck	1	no	W	32	Specifies the tag group number for a tag group with a stalled DMA list command. Writing this tag value to MFC_WrListStallAck restarts (acknowledges) the stalled DMA list command in the tag group.
27	MFC_RdAtomicStat	1	yes	R	32	Returns the status of the most-recently completed immediate MFC atomic-update command (the immediate MFC atomic-update commands are getllar, putllc, or putlluc).

Table 19-14. MFC Channels for Event Monitoring and Management

SPE Channel #	Mnemonic	Max. Entries	Blocking	R/W	Width (bits)	Functions
0	SPU_RdEventStat	1	yes	R	32	Returns events that are both enabled and pending.
1	SPU_WrEventMask	1	no	W	32	Specifies which pending events will affect the value returned by a read of the SPU_RdEventStat channel.
2	SPU_WrEventAck	1	no	W	32	Specifies the pending events to be acknowledged (cleared).

19.4.3 DMA Command Issue Status and Errors

The progress of SPU-initiated DMA commands can be monitored in various ways including checking the completion status of tag groups and checking the number of entries available in the MFC SPU Command Queue. SPU-initiated DMA commands can also fail due to exceptional conditions that include address-translation faults, command errors, and alignment errors. These exceptional conditions are detected and handled by privileged software executing on the PPE.

19.4.3.1 *DMA Tag-Group Completion*

Each MFC command is tagged with a 5-bit Tag-Group ID. The same identifier may be used for multiple MFC commands. The set of all commands with the same identifier is defined as a tag group. Software can use this tag-group identifier to check the completion status of all queued commands in one or more tag groups. When one or more tag group complete, an interrupt may be raised in the SPU, if enabled by software.

Each of the 32 tag groups is assigned a bit in the 32-bit `MFC_WrTagMask` register. When software needs to check the completion status of one or more tag groups, it sets the `MFC_WrTagMask` register with a value that has the corresponding bit set for each DMA tag group of interest. Tag group 31 is assigned the most-significant bit of `MFC_WrTagMask`, and tag group 0 is assigned the least-significant bit.

Software can use three basic protocols to determine the completion status of one or more commands using tag groups:

- Poll the tag-group update status.
- Wait for tag-group update event.
- Enable interrupt on tag-group update event and process other tasks.

These protocols are discussed in the following sections.

Poll Tag-Group Update Status

The basic procedure for polling the completion of an MFC command or group of commands is:

1. Set the MFC Write Tag-Group Query Mask, `MFC_WrTagMask`.
2. Write ‘0’ to `MFC_WrTagUpdate` to request immediate tag status update.
3. Read `MFC_RdTagStat` to find out if the tag groups have completed.
4. Repeat steps 2 and 3 as necessary.

The following assembler-language program fragment illustrates the above procedure.

```
# Inputs:
#   $0 contains tag mask
# Outputs:
#   $1 contains completed tag status

# Set tag group mask. Once written, the tag mask is retained
# until changed by another write to the tag mask.
    wrch $MFC_WrTagMask, $0

# Set up for immediate tag status update.
    il $1, 0

repeat:
    wrch $MFC_WrTagUpdate, $1
    rdch $1, $MFC_RdTagStat
    brz $1, repeat
```

The following C-language program fragment performs the same function as the assembler-language fragment above. The C-language intrinsic **spu_mfcstat(t)** writes the `t` argument to `MFC_WrTagUpdate` and then returns the value read from `MFC_RdTagStat`.

```
#include <spu_intrinsics.h>
#include <spu_mfcio.h>

unsigned int tag_id = 0;
unsigned int tag_mask = 1 << tag_id;

spu_writech(MFC_WrTagMask, tag_mask);

do {
} while (!spu_mfcstat(MFC_TAG_UPDATE_IMMEDIATE)); /* poll for immediate update */
```

Wait for Tag-Group Update

The basic procedure for waiting on a conditional tag event (one or more tag-group completions) is:

1. Set the MFC Write Tag-Group Query Mask, `MFC_WrTagMask`.
2. Write one (`MFC_TAG_UPDATE_ANY`) to `MFC_WrTagUpdate` to request completion status for any tag in the mask; write two (`MFC_TAG_UPDATE_ALL`) if requesting completion status for all tags in the mask.
3. If only waiting for conditional update, read `MFC_RdTagStat`; SPU will stall until the requested completion status is true.
4. If waiting for any one of multiple events, unmask the MFC Tag-Group Status Update event in `SPU_WrEventMask` and read `SPU_RdEventStat`; the SPU will stall until an event is raised. See *Section 18* on page 463 for additional methods of detecting and handling SPU external events.

The following assembler-language program fragment illustrates the above procedure.

```
# Inputs:
#   $0 contains tag mask
# Outputs:
#   $1 contains completed tag status

# Set tag group mask
    wrch $MFC_WrTagMask, $0

# 0x1 for any tag, 0x2 for all tags.
    il $1, 0x1

# Wait for conditional tag status update (stall the SPU).
    wrch $MFC_WrTagUpdate, $1
    rdch $1, $MFC_RdTagStat
```

Note: *Waiting for any tag-group completion with a 0 tag mask can result in a hang or deadlock condition.*

The following C-language program fragment performs the same function as the assembler-language fragment above. The C-language intrinsic **spu_mfcstat(t)** writes the `t` argument to `MFC_WrTagUpdate` and then returns the value read from `MFC_RdTagStat`. If the tag status is not ready, the SPU stalls on the read of `MFC_RdTagStat`.

```
#include <spu_intrinsics.h>
#include <spu_mfcio.h>

unsigned int tag_id = 0;
unsigned int tag_mask = 1 << tag_id;

spu_writech(MFC_WrTagMask, tag_mask);

/* Wait for all ids in tag group to complete (stall the SPU) */
spu_mfcstat(MFC_TAG_UPDATE_ALL);
```

19.4.3.2 *MFC SPU Command Queue Space*

Each MFC has a 16-entry MFC SPU Command Queue for commands initiated by the SPE. As SPE software submits MFC commands, the queue may become full. When the queue is full, SPE software must wait for space to become available before subsequent commands can be issued.

Software can use three basic protocols to wait for space to become available in the MFC SPU Command Queue:

- Poll for queue space.
- Wait for (stall until) queue space is available.
- Wait for (stall until) an external event signals space is available.

Poll for Queue Space

The basic procedure for polling for available space in the command queue is:

1. Read the channel count for `MFC_Cmd`.
2. Repeat until the desired amount of queue space is available.

The following assembler-language example illustrates the above procedure.

```
# Outputs:
#   $0 contains available DMA queue space.

repeat:
    rchcnt $0, $MFC_Cmd
    brz $0, repeat
```

The following C-language program fragment performs the same function as the assembler-language fragment above.

```
#include <spu_intrinsics.h>
#include <spu_mfcio.h>

do {
} while (!spu_readchcnt(MFC_Cmd));
```

Wait for Queue Space

The basic procedure for waiting for available space in the command queue is:

1. If only waiting for space in the MFC SPU Command Queue, simply enqueue the MFC command as usual. The SPU will block (stall) on the channel write to `MFC_Cmd` until space becomes available.
2. If waiting for any one of multiple events, enable the MFC SPU Command Queue Available event in `SPU_WrEventMask` and read from `SPU_RdEventStat`. The SPU will stall until an event is raised.

When SPE software has no independent task to perform until SPU Command Queue space is available, the assembler and C language code sequences are the same as for regular MFC command issue. When software is waiting on SPU Command Queue space and other events, it reads from `SPU_RdEventStat` and stalls until an event is raised. Software then checks the events that are pending and enabled; if SPU Command Queue space is available, software can enqueue more MFC commands.

If SPE software has independent tasks to perform while waiting for queue space to become available, software can enable interrupts for asynchronous event handling. See *Section 18* on page 463 for a complete discussion of event handling and SPU interrupts.

19.4.4 DMA List Command Example

Section 19.2.2 on page 509 summarizes the basics of DMA lists. A DMA list is a sequence of *list elements* that, together with an initiating DMA list command, specifies a sequence of DMA transfers between a single area of LS and possibly discontinuous areas in main storage. Such lists are stored in an SPE's LS, and the sequence of transfers is initiated with a DMA list command such as **getl** or **putl**. DMA list commands can only be issued by programs running on an SPE, but the PPE or other devices can create and store the lists in an SPE's LS. DMA lists can be used to implement scatter-gather functions between main storage and the LS.

Each DMA transfer specified by a list element can transfer up to 16 KB of data, and the list can have up to 2,048 (2 K) list elements. Each list element contains a transfer size, the low half of an effective address, and a stall-and-notify bit which can be used to suspend list execution after transferring a list element whose stall-and-notify bit is set.

The DMA list command specifies one LS starting address for the entire DMA list, and the data in the LS is accessed sequentially with a minimum step of one quadword. Each list element specifies a new starting address in the effective-address space, but the LS address for a list element starts where the last list element left off. The one exception to this rule is that when a list element in a DMA list contains a short transfer size of 1, 2, 4, or 8 bytes, the LS address increments as

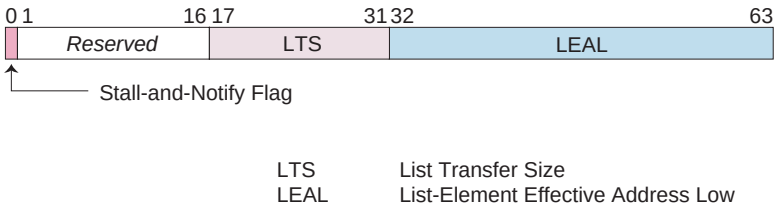
needed to start the next list element on a quadword boundary; thus, for a short transfer size, some data in LS will be skipped between list elements. For transfer sizes that are a multiple of a quadword, no data is skipped.

19.4.4.1 *Creating the List*

Software creates the list and stores it in the LS. Lists must be stored in the LS on an 8-byte boundary. The form of a list element is *{LTS, LEAL}*. The first word (LTS) is the list transfer size, the most-significant bit of which serves as an optional *stall-and-notify* flag. The second word (LEAL) is the list element's 32-bit low-order EA. List elements are processed sequentially, in the order they are stored. If the stall-and-notify flag is set for a list element, the MFC will stop processing the DMA list after performing the transfer for that element until the SPE program clears the DMA List Command Stall-And-Notify Event from the SPU Read Event Status Channel. This gives programs an opportunity to modify subsequent list elements before they are processed by the MFC.

Figure 19-1 shows the format of a DMA list element.

Figure 19-1. DMA List Element



19.4.4.2 *Initiating the Transfers Specified in the List*

After the list is stored in the LS, the execution of the list is initiated by a DMA list command, such as **getl** or **putl**, from the SPE whose LS contains the list. DMA list commands, like single-transfer DMA commands, require that parameters be written to the MFC Command Parameter channels in the manner described in *Section 19.4.1* on page 521. However, a DMA list command requires two different types of parameters then those required by a single-transfer DMA command:

- *MFC_EAL:* This parameter must be written with the *starting Local Store Address (LSA) of the list*, rather then with the EAL. (The EAL is specified in each list element, although it is named List-Element Effective Address Low, LEAL, as shown in *Figure 19-1*.)
- *MFC_Size:* This parameter must be written with the *size of the list*, in bytes, rather then the transfer size. (The transfer size is specified in each list element as the List Transfer Size, LTS, as shown in *Figure 19-1*.) The list size is equal to the number of list elements, multiplied by the size of the transfer-element structure (8 bytes).

The starting LSA and the EA-high (EAH) are specified only once, in the DMA list command that initiates the transfers. The LSA is internally incremented based on the amount of data transferred by each list element. However, if the starting LSA for each list element does not begin on a 16-byte boundary, then hardware automatically increments the LSA to the next 16-byte boundary.

For list elements smaller than 16 bytes, the least-significant four bits of the transfer-element LSA is equivalent to the least-significant four bits of its EAL.

The EAL for each list element is in the 4-GB area defined by EAH. Although each EAL starting address is in a single 4-GB area, individual transfers may cross the 4-GB boundary.

19.4.4.3 *Programming Example*

This C-language sample program creates a DMA list and, in the last line, uses an **spu_mfcdma32** intrinsic to issue a single DMA list command (**getl**) to transfer a main-storage region into LS.

```
/* dma_list_sample.c - SPU MFC-DMA list sample code.
 *
 * This sample defines a list-element data structure, which
 * contains the element's transfer size and low-order 32 bytes of the effective
 * address. Also defined in the structure, but not used by this sample,
 * is the DMA list stall-and-notify bit, which can be used to indicate
 * that the MFC should suspend list execution after transferring a list
 * element whose stall-and-notify bit is set.
 */

#include <spu_mfcio.h>

struct dma_list_elem {
    union {
    unsigned int all32;
    struct {
        unsigned int stall    : 1;
        unsigned int reserved : 15;
        unsigned int nbytes   : 16;
        } bits;
    } size;
    unsigned int ea_low;
};

struct dma_list_elem list[16] __attribute__ ((aligned (8)));

void get_large_region(void *dst, unsigned int ea_low, unsigned int nbytes)
{
    unsigned int i = 0;
    unsigned int tagid = 0;
    unsigned int listsize;

    /* get_large_region
     *    Use a single DMA list command request to transfer
     *    a "large" memory region into LS. The total size to
     *    be copied may be larger than the MFC's single element
     *    transfer limit of 16kb.
     */

    if (!nbytes)
    return;
```

```
    while (nbytes > 0) {
       unsigned int sz;

       sz = (nbytes < 16384) ? nbytes : 16384;
       list[i].size.all32 = sz;
       list[i].ea_low = ea_low;

       nbytes -= sz;
       ea_low += sz;
       i++;
    }


/* Specify the list size and initiate the list transfer */

    listsize = i * sizeof(struct dma_list_elem);
    spu_mfcdma32((volatile *)dst, (unsigned int) &list[0], listsize, tagid,
        MFC_GETL_CMD);
}
```

19.5 Performance Guidelines for MFC Commands

- Minimize small transfers. Transfers of less than one cache line consume bus bandwidth equivalent to a full cache-line transfer.
- Align source and destination addresses of large transfers (128 bytes of larger) to a cache-line boundary.
- Have SPEs Initiate DMA transfers. Let the SPEs pull their data instead of PPE pushing the data to the SPE. This is beneficial for four reasons:
  a. There are 8 times more SPEs than PPEs.
  b. An MFC SPU Command Queue is twice as deep (16 entries) as the PPE's MFC Proxy Command Queue (eight entries).
  c. Consumer-managed transfers are easier to synchronize.
  d. The number of cycles required to initiate a DMA transfer from the SPE is smaller than the number to initiate from the PPE.
- Avoid PPE pre-accesses to large data sets, so that most SPE-initiated DMA transfers come from main storage rather than the PPE's L2 cache. DMA transfers from main storage have high bandwidth with moderate latency, whereas transfers from the L2 have moderate bandwidth with low latency.
- Minimize the use of synchronizing and data-ordering commands.

19.6 Mailboxes

Mailboxes support the sending and buffering of 32-bit messages between an SPE and other devices, such as the PPE and other SPEs. Each SPE can access three mailbox channels, each of which is connected to a mailbox register in the SPU's MFC. Two one-entry mailbox channels—the SPU Write Outbound Mailbox and the SPU Write Outbound Interrupt Mailbox—are provided for sending messages from the SPE to the PPE or other device. One four-entry mailbox channel—the SPU Read Inbound Mailbox—is provided for sending messages from the PPE, or other SPEs or devices, to the SPE. Each of the two outbound mailbox channels has a corresponding MMIO register that can be accessed by the PPE or other devices.

Table 19-15 through *Table 19-17* gives details about the mailbox channels and their associated MMIO registers.

Table 19-15. Mailbox Channels and MMIO Registers

SPE Channel #	Name	Channel Interface					MMIO Register Interface				
		Mnemonic	Max. Entries	Blocking	R/W	Width (bits)	Offset From Base	Mnemonic	Max. Entries	R/W	Width (bits)
28	SPU Write Outbound Mailbox	SPU_WrOutMbox	1	yes	W	32	X'04004'	SPU_Out_Mbox	1	R	32
29	SPU Read Inbound Mailbox	SPU_RdInMbox	4	yes	R	32	X'0400C'	SPU_In_Mbox	4	W	32
30	SPU Write Outbound Interrupt Mailbox[1]	SPU_WrOutIntrMbox	1	yes	W	32	X'04000'	SPU_Out_Intr_Mbox	1	R	64
—	SPU Mailbox Status	—	—	—	—	—	X'04014'	SPU_Mbox_Stat	1	R	32

1. Access to this MMIO register is available only to privileged PPE software.

Table 19-16. Functions of Mailbox Channels

Channel Interface	SPU Read or Write	Functions
SPU_WrOutMbox	W	Writes message data to the outbound mailbox.
SPU_RdInMbox	R	Returns the next message data from the inbound mailbox
SPU_WrOutIntrMbox	W	Writes message data to the outbound interrupt mailbox.

Table 19-17. Functions of Mailbox MMIO Registers

MMIO Register	PPE Read or Write	Functions
SPU_Out_Mbox	R	Returns the message data from the corresponding SPU outbound mailbox.
SPU_In_Mbox	W	Writes message data to the SPU inbound mailbox.
SPU_Out_Intr_Mbox[1]	R	Returns the message data from the corresponding SPU outbound interrupt mailbox.
SPU_Mbox_Stat	R	Returns the number of available mailbox entries.

1. Access to the SPU_Out_Intr_Mbox MMIO register is available only to privileged PPE software.

19.6.1 Reading and Writing Mailboxes

Data written by an SPE program to one of the outgoing mailboxes using an SPE write-channel (**wrch**) instruction is available to any processor element or device that reads the corresponding MMIO register in the main-storage space. Data written by a device to the SPU Read Inbound Mailbox using an MMIO write is available to an SPE program by reading that mailbox with a read-channel (**rdch**) instruction.

An MMIO read from either of the outbound mailboxes, or a write to the inbound mailbox, can be programmed to raise an SPE event, which in turn, may cause an SPU interrupt. A **wrch** instruction to the SPU Write Outbound Interrupt Mailbox can also be programmed to cause an interrupt to the PPE or other device, depending on interrupt routing (see *Section 9.6 Direct External Interrupts* on page 262 and *Section 9.8 SPU and MFC Interrupts Routed to the PPE* on page 276).

Each time a PPE program writes to the four-entry inbound mailbox, the channel count for the SPU Read Inbound Mailbox channel increments. Each time an SPE program reads the mailbox, the channel count decrements. The inbound mailbox acts as a FIFO queue; SPE software reads the oldest data first. If the PPE program writes more than four times before the SPE program reads the data, then the channel count stays at ‘4’, and the fourth location contains the last data written by the PPE. For example, if the PPE program writes five times before the SPE program reads the data, then the data read is the first, second, third, and fifth messages that were written. The fourth message that was written is lost.

19.6.2 Mailbox Blocking

Mailbox operations are blocking operations for an SPE: an SPE write to an outbound mailbox that is already full stalls the SPE until a entry is cleared in the mailbox by a PPE read. Similarly, a **rdch** instruction—but not a read-channel-count (**rchcnt**) instruction—from an empty inbound mailbox is stalled until the PPE writes to the mailbox. That is, if the channel count is ‘0’ for a blocking channel, then an **rdch** or **wrch** instruction for that channel causes the SPE to stall until the channel count changes from ‘0’ to non-zero.

To prevent stalling, SPE software should read the channel count associated with the mailbox before deciding whether to read or write the mailbox channel. This stalling behavior for the SPE does not apply to the PPE; if the PPE sends a message to the inbound mailbox and the mailbox is full, the PPE will not stall.

Since a **wrch** instruction will stall when it tries to send a value to a full outbound mailbox, SPE software cannot over-run an outbound mailbox. All outbound messages must be read by some entity outside the SPU before space is made available for more outbound messages. In contrast, the SPU Read Inbound Mailbox can be over-run by an outside entity because the MMIO write used for this purpose does not stall. When the PPE or other device writes to a full inbound mailbox, the last value written to it is overwritten and is lost. See the code sample in *Section 19.7.6.1* on page 545 for a safeguard to avoid this situation.

19.6.3 Dealing with Anticipated Messages

There are at least three ways to deal with anticipated mailbox messages:

- SPE software uses an **rdch** instruction to read the inbound mailbox channel, which will block until a mailbox message arrives.

- SPE software uses a **rchcnt** instruction on the mailbox channel, which will return the count ('0' or non-zero). If the count is '0', software can work on other tasks and check again later.
- SPE software enables interrupts to respond to mailbox events. Software can work on other tasks and need never check because the mailbox event interrupt handlers will be activated when mailbox status changes.

19.6.4 Uses of Mailboxes

Mailbox message values are intended to communicate messages up to 32 bits in length, such as buffer completion flags or program status. In fact, however, they can be used for any short-data-transfer purpose, such as sending of storage addresses, function parameters, command parameters, and state-machine parameters.

Mailboxes are useful, for example, when the SPE places computational results in main storage via DMA. After requesting the DMA transfer, the SPE waits for the DMA transfer to complete and then writes to an outbound mailbox to notify the PPE that its computation is complete.

If the SPE sends a mailbox message after waiting for a DMA transfer to complete, this ensures only that the SPE's LS buffers are available for reuse. It does not guarantee that data has been coherently written to main storage. The SPE could solve this problem by issuing an **mfcsync** command prior to notifying the PPE. But doing so would be inefficient. Instead, the preferred method is to have the PPE receive the notification and then issue an **lwsync** instruction before accessing any of the resulting data.

Alternatively, an SPE can notify the PPE that it has completed computation by writing, via DMA, such a notification to main storage, from which the PPE can read the notification. (This is sometimes referred to as a *writeback DMA command*, although no such DMA command is defined.) In this case, the data and the writeback must be ordered. To ensure ordering, an `mfceieio` command must be issued between the data DMA commands and the notification to main storage.

Although the mailboxes are primarily intended for communication between the PPE and the SPEs, they can also be used for communication between an SPE and other SPEs, processors, or devices. For this to happen, privileged software needs to allow one SPE to access the mailbox register in another SPE by mapping the target SPE's problem-state area into the EA space of the source SPE. If software does not allow this, then only atomic operations and signal notifications are available for SPE-to-SPE communication.

19.6.5 SPU Outbound Mailboxes

The MFC provides two one-entry mailbox channels—the SPU Write Outbound Mailbox and the SPU Write Outbound Interrupt Mailbox—for sending messages from the SPE to the PPE or other device that can read from the main-storage space.

19.6.5.1 *SPU Write Outbound Mailbox Channel*

The value written to the SPU Write Outbound Mailbox channel (`SPU_WrOutMbox`) is entered into the outbound mailbox in the MFC if the mailbox has capacity to accept the value. If the mailbox can accept the value, the channel count for `SPU_WrOutMbox` is decremented by '1'.

If the outbound mailbox is full, the channel count will read as ‘0’. If SPE software writes a value to `SPU_WrOutMbox` when the channel count is ‘0’, the SPU will stall on the write. The SPU remains stalled until the PPE or other device reads a message from the outbound mailbox by reading the MMIO address of the mailbox. When the mailbox is read through the MMIO address, the channel count is incremented by ‘1’.

19.6.5.2 *SPU Write Outbound Interrupt Mailbox Channel*

The value written to the SPU Write Outbound Interrupt Mailbox channel (`SPU_WrOutIntrMbox`) is entered into the outbound interrupt mailbox if the mailbox has capacity to accept the value. If the mailbox can accept the message, the channel count for `SPU_WrOutIntrMbox` is decremented by ‘1’, and an interrupt is raised in the PPE or other device, depending on interrupt enabling and routing. There is no ordering of the interrupt and previously issued MFC commands.

The raising of an interrupt depends on privileged PPE software having enabled the interrupt for this channel, as described in *Section 9.6 Direct External Interrupts* on page 262 and *Section 9.8 SPU and MFC Interrupts Routed to the PPE* on page 276. After handling the PPE-side interrupt, if enabled, privileged PPE software performs an MMIO read of the SPU Write Outbound Interrupt Mailbox register to get the value written by the SPU. In response to the MMIO read, the channel count is incremented by ‘1’. This is the simplest use of the SPU Write Outbound Interrupt Mailbox.

If the outbound interrupt mailbox is full, the channel count will read as ‘0’. If SPE software writes a value to `SPU_WrOutIntrMbox` when the channel count is ‘0’, the SPU will stall on the write. The SPU remains stalled until the PPE or other device reads a mailbox message from the outbound interrupt mailbox by reading the MMIO address of the mailbox. When this is done, the channel count is incremented by ‘1’.

The field assignments and definitions for SPU Write Outbound Interrupt Mailbox are identical to those for the SPU Write Outbound Mailbox, shown in *Section 19.6.5.1* on page 533.

19.6.5.3 *SPU Side*

Writing SPU Write Outbound Mailbox Data

SPE software can write to the SPU Write Outbound Mailbox channel to put a mailbox message in the SPU Write Outbound Mailbox. This write-channel instruction will return immediately if there is sufficient space in the SPU Write Outbound Mailbox Queue to hold the message value. If there is insufficient space, the write-channel instruction will stall the SPU until the PPE reads from this mailbox.

The same behavior applies to the SPU Write Outbound Interrupt Mailbox on the SPU side, the only difference is the channel number.

The following assembler-language program fragment shows how to write to an SPU Write Outbound Mailbox.

```
# To write to the SPU Write Outbound Interrupt Mailbox
# instead of the SPU Write Outbound Mailbox, simply replace
# SPU_WrOutMbox with SPU_WrOutIntrMbox in the
# following example.
```

```
# Input:
#        $1 contains the mailbox value to be written

wrch $SPU_WrOutMbox, $1
```

The following C-language program fragment shows how to write to the SPU Write Outbound Mailbox.

```
unsigned int mb_value;

spu_writech(SPU_WrOutMbox, mb_value);
```

Waiting to Write SPU Write Outbound Mailbox Data

To avoid an SPU stall condition, the SPU can use the read-channel-count instruction on the SPU Write Outbound Mailbox channel to determine if the queue is empty before writing to the channel. If the read-channel-count instruction returns '0', the SPU Write Outbound Mailbox Queue is full. If the read channel-count instruction returns a non-zero value, the value indicates the number of free entries in the SPU Write Outbound Mailbox Queue. When the queue has free entries, the SPU can write to this channel without stalling the SPU.

The same behavior described above also applies to the SPU Write Outbound Interrupt Mailbox Queue.

The following assembler-language program fragment shows how to poll SPU Write Outbound Mailbox or SPU Write Outbound Interrupt Mailbox.

```
# To write to the SPU Write Outbound Interrupt Mailbox
# instead of the SPU Write Outbound Mailbox, simply replace
# SPU_WrOutMbox with SPU_WrOutIntrMbox in the
# following example.
# Input:
#        $1 contains the mailbox value to be written

repeat:
    rchcnt $2, $SPU_WrOutMbox
    brz $2, repeat

    wrch $SPU_WrOutMbox, $1
```

The following C-language program fragment shows how to poll SPU Write Outbound Mailbox or SPU Write Outbound Interrupt Mailbox.

```
/*
 * To write the value 1 to the SPU Write Outbound Interrupt Mailbox instead
 * of the SPU Write Outbound Mailbox, simply replace SPU_WrOutMbox
 * with SPU_WrOutIntrMbox in the following example.
 */
```

```
unsigned int mb_value;

do {
    /*
     * Do other useful work while waiting.
     */
} while (!spu_readchcnt(SPU_WrOutMbox));

spu_writech(SPU_WrOutMbox, mb_value);
```

The SPU can also avoid a stall by enabling the SPU Event Facility, described in *Section 18* on page 463. Using events involves the following steps:

1. Enable the PPE Core Mailbox Available event or the PPE Core Interrupt Mailbox Available event in SPU Write Event Mask.
2. Monitor for this event. Software can poll with a read-channel-count instruction on SPU Read Event Status; software can block (stall) with a read-channel instruction on the SPU Read Event Status Channel; software can enable SPU interrupts and use interrupt handlers to service mailbox events as they are encountered asynchronously.
3. When the event is detected, it must be acknowledged by writing the corresponding bit to the SPU Write Event Acknowledgement channel. A typical handler for this event would perform the acknowledge and also write a value to SPU Write Outbound Mailbox or SPU Write Outbound Interrupt Mailbox.

The following assembler-language program fragment shows how to poll for or block on an SPU Write Outbound Mailbox available event. To block, instead of poll, simply delete the repeat loop below. To write to the SPU Write Outbound Interrupt Mailbox instead of the SPU Write Outbound Mailbox, simply replace `SPU_WrOutMbox` with `SPU_WrOutIntrMbox` and use the appropriate event bit for SPU Write Outbound Interrupt Mailbox in the following example.

```
# Inputs:
#   $2 contains SPU Write Outbound Mailbox event bit
#   $5 contains the 32-bit value to be written to the SPU Write Outbound Mailbox

    il   $2, 0x80
    wrch $SPU_WrEventMask, $2           # Write event mask

repeat:
    rchcnt $3, $SPU_RdEventStat
    brz $3, repeat                      # Wait for event to occur

    rdch $4, $SPU_RdEventStat           # Get event bit
    wrch $SPU_WrEventAck, $2            # Acknowledge event

    wrch $SPU_WrOutMbox, $5             # Send the mailbox message
```

The following C-language program fragment shows how to poll for or block on an SPU Write Outbound Mailbox Available event. To block, instead of poll, simply delete the do/while loop below. To write to the SPU Write Outbound Interrupt Mailbox instead of the SPU Write Outbound Mailbox, simply replace `SPU_WrOutMbox` with `SPU_WrOutIntrMbox` and use the appropriate event bit for SPU Write Outbound Interrupt Mailbox in the following example.

```
#define MBOX_AVAILABLE_EVENT  0x00000080
unsigned int event_status;
unsigned int mb_value;

spu_writech(SPU_WrEventMask, MBOX_AVAILABLE_EVENT);

do {
    /*
     * Do other useful work while waiting.
     */
} while (!spu_readchcnt(SPU_RdEventStat));

event_status = spu_readch(SPU_RdEventStat);          /* read status */

spu_writech(SPU_WrEventAck, MBOX_AVAILABLE_EVENT);   /* acknowledge event */

spu_writech(SPU_WrOutMbox, mb_value);                /* send mailbox message */
```

Note: *The example above does not fully accommodate the occurrence of a phantom event. For details on handling phantom events, see Section 18.6.1 on page 473.*

19.6.5.4 *PPE Side*

Before PPE software can read data from one of the SPU Write Outbound Mailboxes, it must first read the Mailbox Status Register to determine that unread data is present in the SPU Write Outbound Mailbox or SPU Write Outbound Interrupt Mailbox; otherwise, stale or undefined data may be returned.

To determine that unread data is available in the SPU Write Outbound Mailbox, PPE software reads the Mailbox Status register and extracts the count value from the `SPU_Out_Mbox_Count` field. If the count is non-zero, then at least one unread value is present. If the count is ‘0’, PPE software should not read the SPU Write Outbound Mailbox Register because it will get incorrect data and should poll the Mailbox Status register.

The following C-language example shows how PPE software can read from the SPU Write Outbound Mailbox of an SPE. The example assumes the existence of a operating system call, like **get_ps**, that returns a base address for the problem state area, and some defines, like `SPU_Mbox_Stat`, and `SPU_Out_Mbox`, that specify offsets into the problem state area corresponding to the `SPU_Mbox_Stat` and `SPU_Out_Mbox` registers, respectively.

```
void *ps = get_ps();
unsigned int mb_status;
unsigned int new;
unsigned int mb_value;
```

```
do {
    mb_status = *((volatile unsigned int *)(ps + SPU_Mbox_Stat));
    new = mb_status & 0x000000FF;
} while ( new == 0 );
/*
 * Issue an eieio instruction to ensure that the last
 * Mailbox Status Register read is performed prior to the first
 * SPU Write Outbound Mailbox Register read.
 */
__asm__("eieio");

mb_value = *((volatile unsigned int *)(ps + SPU_Out_Mbox));
```

SPE software typically uses the SPU Write Outbound Interrupt Mailbox when PPE software needs to be interrupted to detect mailbox data from an SPU. For details on using privileged operations to enable, process, and reset the PPE interrupt associated with the SPU Write Outbound Interrupt Mailbox, see *Section 9 PPE Interrupts* on page 237.

As for the SPU Write Outbound Mailbox, a non-zero value in the `SPU_Out_Intr_Mbox_Count` field of the Mailbox Status Register indicates that unread data is present in the SPU Write Outbound Interrupt Mailbox. The same code example provided above for the SPU Write Outbound Mailbox can be used for reading the SPU Write Outbound Interrupt Mailbox, after changes have been made for the location of the `SPU_Out_Intr_Mbox_Count` field in the Mailbox Status Register and the appropriate offset is used for the SPU Write Outbound Interrupt Mailbox Register.

19.6.6 SPU Inbound Mailbox

The MFC provides one mailbox for a PPE to send information to an SPU: the SPU Read Inbound Mailbox. This mailbox has four entries; that is, the PPE can have up to four 32-bit messages pending at a time in the SPU Read Inbound Mailbox.

19.6.6.1 *SPU Read Inbound Mailbox Channel*

If the SPU Read Inbound Mailbox channel (`SPU_RdInMbox`) has a message, the value read from the mailbox is the oldest message written to the mailbox. If the inbound mailbox is empty, the `SPU_RdInMbox` channel count will read as ‘0’.

If SPE software reads from SPU_RdInMbox when the channel count is ‘0’, the SPU will stall on the read. The SPU remains stalled until the PPE or other device writes a message to the mailbox by writing to the MMIO address of the mailbox.

When the mailbox is written through the MMIO address, the channel count is incremented by ‘1’. When the mailbox is read by the SPU, the channel count is decremented by '1'.

19.6.6.2 *PPE Side*

An important difference between the SPU Read Inbound Mailbox and the SPU Write Outbound Mailboxes is that the SPU Read Inbound Mailbox can be overrun by a PPE. A PPE writing to the SPU Read Inbound Mailbox will not stall when this mailbox is full. When a PPE overruns the SPU Read Inbound Mailbox, mailbox message data will be lost.

There is a simple mechanism for the PPE to use to avoid overruns. The fields of the SPU Mailbox Status register, `SPU_Mbox_Stat`, shown in *Table 19-18*, contains an `SPU_In_Mbox_Count` field that contains the number of available entries in the SPU Read Inbound Mailbox. To avoid overruns, the PPE should read the `SPU_Mbox_Stat` register and send no more mailbox values than indicated in the `SPU_In_Mbox_Count` field.

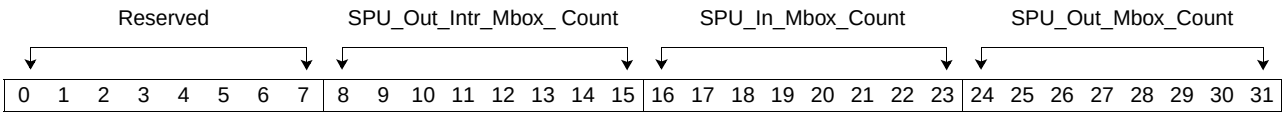


Table 19-18. Fields of Mailbox Status Register (SPU_Mbox_Stat)

Bits	Field Name	Description
0:7	Reserved	Set to zeros.
8:15	SPU_Out_Intr_Mbox_ Count	Number of valid entries in the SPU Write Outbound Interrupt Mailbox.
16:23	SPU_In_Mbox_Count	Number of available entries in the SPU Read Inbound Mailbox.
24:31	SPU_Out_Mbox_Count	Number of valid entries in the SPU Write Outbound Mailbox.

The following C-language program fragment shows how to write four 32-bit messages to the SPU Read Inbound Mailbox of a particular SPU from the PPE. The code avoids overrunning the SPU Read Inbound Mailbox. The example assumes the existence of a operating system call, like **get_ps**, that returns a base address for the problem state area, and some defines, like `SPU_Mbox_Stat`, and `SPU_In_Mbox`, that specify offsets into the problem state area corresponding to the `SPU_Mbox_Stat` and `SPU_In_Mbox` registers, respectively.

```
void *ps = get_ps();
unsigned int j,k = 0;
unsigned int mb_status;
unsigned int slots;
unsigned int mb_value[4] = {0x1, 0x2, 0x3, 0x4};
do {
    /*
     * Poll the Mailbox Status Register until the SPU_In_Mbox_Count
     * field indicates there is at least one slot available in the
     * SPU Read Inbound Mailbox.
     */
    do {
        mb_status = *((volatile unsigned int *)(ps + SPU_Mbox_Stat));
        slots = (mb_status & 0x0000FF00) >> 8;
    } while ( slots == 0 );
    /*
     * Issue an eieio instruction to ensure that the last
     * Mailbox Status Register read is performed prior to the first
```

```
     * SPU Read Inbound Mailbox Register write.
     */
    __asm__("eieio");

    for (j=0; j<slots && k < 4; j++) {
        *((volatile unsigned int *)(ps + SPU_In_Mbox)) = mb_value[k++];
    }
} while ( k < 4 );
```

19.6.6.3 ***SPU Side***

Reading SPU Read Inbound Mailbox Data

SPE software can use a read-channel instruction on the SPU Read Inbound Mailbox channel to read the contents of its SPU Read Inbound Mailbox. This channel read will return immediately if any data written by the PPE is waiting in the SPU Read Inbound Mailbox. This read-channel instruction will cause the SPU to stall if the SPU Read Inbound Mailbox is empty.

The following assembler-language program fragment shows how to read the SPU Read Inbound Mailbox:

```
# Outputs:
#        $1 contains the mailbox value read
rdch $1, $SPU_RdInMbox
```

The following C-language program fragment also shows how to read the SPU Read Inbound Mailbox. We assume the `spu_` macro names have been defined previously.

```
unsigned int mb_value;

mb_value = spu_readch(SPU_RdInMbox);
```

Waiting for SPU Read Inbound Mailbox Data

To avoid an SPU stall condition, SPE software can use the read-channel-count instruction on the SPU Read Inbound Mailbox channel to determine if the mailbox is empty. If the read-channel-count instruction returns a ‘0’, the mailbox is empty and a read-channel instruction would stall. If the read channel-count instruction returns a non-zero value, the value indicates the number of waiting mailbox message values in the SPU Read Inbound Mailbox that can be read without stalling the SPE.

The following assembler-language program fragment shows how to poll an SPU Read Inbound Mailbox read channel.

```
# Outputs:
#  $2 contains the SPU Read Inbound Mailbox value
```

```
repeat:
   rchcnt $1, $SPU_RdInMbox
   brz $1, repeat

rdch $2, $SPU_RDInMbox
```

The following C-language program fragment shows how to poll an SPU Read Inbound Mailbox read channel.

```
unsigned int mb_value;

do {
    /*
     * Do other useful work while waiting.
     */
} while (!spu_readchcnt(SPU_RdInMbox));

mb_value = spu_readch(SPU_RdInMbox);
```

The SPU can also avoid a stall by enabling the SPU Event Facility, described in *Section 18* on page 463. Using events involves the following steps:

1. Enable the SPU Inbound Mailbox Available event in SPU Write Event Mask.
2. Monitor for this event. Software can poll with a read-channel-count instruction on SPU Read Event Status; software can block (stall) with a read-channel instruction on the SPU Read Event Status channel; software can enable SPU interrupts and use interrupt handlers to service mailbox events as they are encountered asynchronously.
3. When the event is detected, it must be acknowledged by writing the corresponding bit to the SPU Write Event Acknowledgement channel. A typical handler for this event would perform the acknowledge and also read one or more inbound mailbox value(s) from the SPU Read Inbound Mailbox channel.

The following assembler-language program fragment shows how to poll or block for an SPU Read Inbound Mailbox available event. To block, instead of poll, simply delete the repeat loop below. Only one mailbox value is read to simplify this example, although more than one may have been indicated by the SPU Read Inbound Mailbox channel count.

```
# Inputs:
#   $2 contains SPU Read Inbound Mailbox event bit
# Outputs:
#   $6 contains one SPU Read Inbound Mailbox value

    il     $2, 0x10
    wrch   $SPU_WrEventMask, $2      # Enable only inbound mailbox event

repeat:
    rchcnt $3, $SPU_RdEventStat
    brz    $3, repeat                # Wait for event
```

```
    rdch   $4, $SPU_RDEventStat      # Read event status
    wrch   $SPU_WrEventAck, $2       # Use the value to acknowledge the event
    rdch   $6, $SPU_RdInMbox         # Read first available message
```

The following C-language program fragment shows how to poll or block for an SPU Read Inbound Mailbox available event. To block, instead of poll, simply delete the do/while loop below. All available mailbox messages are placed in the `mb_value` array, and the `mb_count` is set to indicate the number of messages read.

```
#define MFC_SPU_In_Mbox_Event  0x00000010
unsigned int event_status;
unsigned int mb_count, i;
unsigned int mb_value[4];

spu_writech(SPU_WrEventMask, MFC_SPU_In_Mbox_Event);

do {
   /*
    * Do other useful work while waiting.
    */
} while (!spu_readchcnt(SPU_RdEventStat));

event_status = spu_readch(SPU_RdEventStat);

spu_writech(SPU_WrEventAck, event_status);

mb_count = spu_readchcnt(SPU_RdInMbox);
for (i=0; i< mb_count; i++) {
    mb_value[i] = spu_readch(SPU_RdInMbox);
}
```

19.7 Signal Notification

SPE signal-notification channels are connected to inbound registers (into the SPE). The PPE, other SPEs, and other devices use the signal notification registers to send information, such as a buffer-completion synchronization flag, to an SPE. An SPE has two 32-bit signal-notification registers, each of which has a corresponding MMIO register that can be written with signal-notification data.

When SPE software reads a signal-notification channel, hardware clears the channel atomically. In contrast, a read in the MMIO main-storage space does not clear a signal-notification register. SPE software can use polling or blocking when waiting for a signal to appear, or it can set up interrupts to catch signals as they appear asynchronously.

The PPE sends a signal-notification message to the SPE by writing to a MMIO register in the SPE's MFC. The signal is latched in the MMIO register, and the SPU the signal value by executing a read-channel (**`rdch`**) instruction.

One SPU can send a signal-notification message to another SPU using one of three special MFC commands: **sndsig**, **sndsigf**, and **sndsigb**. All of these commands are implemented in the same manner as a DMA **put** command, with the effective address of an MMIO register as the destination. In fact, a DMA **put** command can be used to perform exactly the same function as a send-signal command; the send-signal commands are defined in the architecture to support possible future CBE implementations that optimize signal-notification performance.

19.7.1 SPU Signalling Channels

There are two SPU signal-notification channels, one to read each of the two signal-notification MMIO registers. A signal can be from one bit to 32 bits long. *Table 19-19* through *Table 19-21* list the signal-notification channels and associated MMIO registers. An SPU read from a signal-notification channel will be stalled when no signal is pending at the time of the read.

Table 19-19. Signal-Notification Channels and MMIO Registers

SPE Channel #	Name	Channel Interface					MMIO Register Interface				
		Mnemonic	Max. Entries	Blocking	R/W	Width (bits)	Offset From Base	Mnemonic	Max. Entries	R/W	Width (bits)
3	SPU Signal Notification 1	SPU_RdSigNotify1	1	yes	R	32	x'1400C'	SPU_Sig_Notify_1	1	R/W	32
4	SPU Signal Notification 2	SPU_RdSigNotify2	1	yes	R	32	x'1C00C'	SPU_Sig_Notify_2	1	R/W	32
—	SPU Configuration	—	—	—	—	—	x'04078'	SPU_Cfg	1	R/W	64

Table 19-20. Functions of Signal-Notification Channels

Channel Interface	SPU Read or Write	Functions
SPU_RdSigNotify1	R	Returns signal-notification data.
SPU_RdSigNotify2	R	Returns signal-notification data.

Table 19-21. Functions of Signal-Notification MMIO Registers

MMIO Register	PPE Read or Write	Functions
SPU_Sig_Notify_1	R/W	Reads or writes signal-notification data.
SPU_Sig_Notify_2	R/W	Reads or writes signal-notification data.
SPU_Cfg[1]	R/W	Reads or writes the configuration (either OR mode or overwrite mode) of the SPU signal notification registers.

1. Access to this MMIO register is available only to privileged PPE software.

The two signal-notification channels each have one entry. Thus, the value returned from a read-channel-count (**`rchcnt`**) instruction indicates the presence or absence of an available signal value; if the channel-count value is '1', a signal is available; if the value is '0', no signal is available. When the channel count is '0' for a signal-notification channel, a read-channel (**`rdch`**) instruction directed at that channel will stall the SPU until a signal is available.

19.7.2 Uses of Signaling

Like mailboxes, signal-notification channels are useful when the SPE places computational results in main storage via DMA. After requesting the DMA transfer, the SPE waits for the DMA transfer to complete and then sends a signal to notify the PPE that its computation is complete. In this case, waiting for the DMA transfer to complete only ensures that the SPE's LS buffers are available for reuse and does not guarantee that data has been coherently written to main storage, as described in *Section 19.6.4 Uses of Mailboxes* on page 533.

19.7.3 Mode Configuration

Signal-notification registers can be configured independently by PPE software to operate in either *OR mode* or *overwrite mode*. The SPU Configuration (`SPU_Cfg`) MMIO register is used for this purpose.

In OR mode, the MFC combines all the values written to a signal-notification register using a logical OR operation until the SPU reads the register through the corresponding channel. When the SPU reads the register, the register is reset to '0'. OR mode allows multiple signals to be accumulated in a single signal-notification register. Each signal producer is assigned a subset of the 32 bits in the signal-notification register; this allows the signal producers to send their signals at any time and independently or other signal producers. When SPE software reads the signal-notification register, it becomes aware of all the signals that have been sent since the most-recent read of the register. This mode of signalling can be referred to as *many-to-one signalling*.

In overwrite mode, each value written to a signal-notification register overwrites the value in register. This mode of signalling can be referred to as *one-to-one signaling*. In the case of one-to-one signaling, there is usually no substantial difference in performance between signaling and using an inbound mailbox.

Since the channel count for signal-notification registers can be at most '1', SPE software cannot determine how many writes to the signal notification register have taken place. If software needs to be made aware immediately of a write to a signal-notification register, it can enable interrupts and configure the event facility to interrupt when signals arrive (see *Section 18* on page 463).

19.7.4 SPU Signal Notification 1 Channel

The value read from the SPU Signal Notification 1 channel (`SPU_RdSigNotify1`) is the 32-bit value of signal register 1. Reading this channel atomically resets to '0' any bits in the corresponding signal register that were set to '1'. A read from this channel when no signals are pending stalls the SPU until a signal is sent. If no signals are pending, a read-channel-count instruction for this channel returns '0'; if unread signals are pending, it returns one.

19.7.5 SPU Signal Notification 2 Channel

The value read from the SPU Signal Notification 2 channel (`SPU_RdSigNotify2`) is the 32-bit value of signal register 2. Reading this channel atomically resets to '0' any bits in the corresponding signal register that were set to '1'. A read from this channel when no signals are pending stalls the SPU until a signal is sent. If no signals are pending, a read-channel-count instruction for this channel returns '0'; if unread signals are pending, it returns one.

19.7.6 Sending Signals

19.7.6.1 *From the PPE*

PPE software and other system devices send a signal to an SPE by performing a 32-bit MMIO write to the effective address of the desired signal-notification register. This write increments the count of the corresponding channel to one.

When the target signal-notification register is in overwrite mode, a PPE application can send multiple signals to an SPU without overrunning the signal facility by using non-zero signal values and verifying that the previous value has been read by the SPU before writing to the signal-notification register. PPE software verifies that the previous value has been read by reading from the MMIO address of the signal-notification register and ensuring that the MFC has reset the value to ‘0’; a value of ‘0’ means SPE software read the previous value.

When the target signal-notification register is in OR mode, a PPE application can send multiple signals to an SPU without overrunning the signal facility by designating a unique bit field in the signal value to represent a particular signaller. Then, in a manner similar to the above, PPE software should verify that the bit field for the signaller is ‘0’ before initiating another signal.

The following C-language program fragment shows how the PPE can write to a signal-notification register of a particular SPE. The code avoids overrunning the signal notification facility. We assume the `SPU_Sig_Notify_1` and `SPU_Sig_Notify_2` macro names, representing offsets into the problem-state region, have been defined previously. This is an example of overwrite-mode signalling, and we assume that privileged PPE software has already configured the SPU Signal Notification 1 register in overwrite mode.

```
void *ps = get_ps();
unsigned int prev_sn_value;
unsigned int sn_value = 0xF0;

do {
    /* Do some other useful work while waiting. */
    prev_sn_value = *((volatile unsigned int *)(ps + SPU_Sig_Notify_1));
} while ( prev_sn_value != 0 );
/*
 * Issue an eieio instruction to ensure that the last
 * Signal Notification Register read is performed prior to the
 * following Signal Notification Register write.
 */
__asm__(“eieio”);

*((volatile unsigned int *)(ps + SPU_Sig_Notify_1)) = sn_value; // Write signal value
```

19.7.6.2 *From an SPE*

SPE software can send a signal to another SPE or some other device by issuing an MFC send-signal command (*Section 19.2.3* on page 510) and specifying the effective address of the target SPU signal-notification channel and a signal value. The command increments the channel count of the target SPU’s signal-notification channel to one.

When the target signal-notification register is in overwrite mode, an SPU application can send multiple signals to a target SPU without overrunning the signal facility by using non-zero signal values and verifying that the previous value has been read by the target SPU before performing an MFC send-signal command. SPE software verifies that the previous value has been read by by performing an MFC **get** command from the effective address of the target SPU signal-notification register and ensuring that it has been reset to ‘0’ by a channel read on the target SPU.

When the target signal-notification register is in OR mode, an SPU application can send multiple signals to another SPU without overrunning the signal facility by designating a unique bit field in the signal value to represent a particular signaller. Then, in a manner similar to above, SPE software should ensure that the bit field for the signaller is ‘0’ before initiating another signal.

Send-Signal Command

The following ABI-compliant assembler-language program fragment shows how a source SPE can write to a signal-notification register in a target SPE using an MFC **sndsig** command. SPE software first performs an MFC **get** command to confirm that the target signal-notification register is ‘0’ before writing a new signal value. This is an overwrite-mode signalling example, and we assume that privileged software has already configured the SPU Signal Notification 1 register in the target SPE to operate in overwrite mode.

```
# Inputs:
#   $3 contains 64 effective address of signal register
#   $4 contains 32-bit value to be written
#   $5 contains tag id, between [0..31]
#

    .set MFC_GET_CMD, 0x40
    .set MFC_SNDSIG_CMD, 0xA0

send_signal:
    ila  $6, 4                  # size of signal register
    ila  $7, signal_word        # LSA of signal word
    rotqbyi $8, $3, 4           # EAL of signal register

    ila  $9, 1                  # initialize tag mask
    shl  $9, $9, $5
    wrch $MFC_WrTagMask, $9

    # Read signal notification register and wait until 0
repeat:
    wrch $MFC_LSA, $7
    wrch $MFC_EAH, $3
    wrch $MFC_EAL, $8
    wrch $MFC_Size, $6
    wrch $MFC_TagID, $5
    ila  $9, MFC_GET_CMD
    wrch $MFC_Cmd, $9           # enqueue command
    ila  $9, 2
    wrch $MFC_WrTagUpdate, $9
```

```
    rdch $9, $MFC_RdTagStat     # wait for DMA get command to complete
    lqa  $9, signal_qword
    rotqbyi $9, $9, 12
    brz  $9, repeat

    # Place the requested data into the LS to be DMA'ed
    rotqbyi $9, $4, -12
    stqa $9, signal
    dsync

    # Write the specified data to the signal notification register
    wrch $MFC_LSA, $7
    wrch $MFC_EAH, $3
    wrch $MFC_EAL, $8
    wrch $MFC_Size, $6
    wrch $MFC_TagID, $5
    ila  $9, MFC_SNDSIG_CMD
    wrch $MFC_Cmd, $9           # enqueue command

    bi    $0                    # return to caller

    # LS data buffer used for DMA operations
    .align 4
signal_qword:
    .skip 12
signal_word:
    .skip 4
```

Here is a C-language subroutine that is comparable to the example above.

```
#include <spu_intrinsics.h>
#include <spu_mfcio.h>

void send_signal(unsigned long long ea, unsigned int value, unsigned int tag)
{
  volatile vector unsigned int signal_qword;
  unsigned int eal, eah;

  spu_writech(MFC_WrTagMask, 1 << tag);

  /* Read signal notification register and wait until 0
   */
  do {
    mfc_get((volatile void *)(&signal_qword) + 12, ea, 4, tag, 0, 0);
    spu_mfcstat(MFC_TAG_UPDATE_ALL);
  } while (spu_extract(signal_qword, 3));

  /* Place the requested data into the LS to be DMA'ed
   */
  signal_qword = spu_promote(value, 3);
```

```
  spu_dsync();

  mfc_sndsig((volatile void *)(&signal_qword) + 12, ea, tag, 0, 0);
}
```

19.7.7 Receiving Signals

19.7.7.1 *Reading Signals*

SPE software uses a read-channel instruction on the signal-notification channel of interest to receive a 32-bit signal value. This read-channel instruction will return immediately, reset any set bits in the signal-notification register, and reset the channel count to ‘0’ if the associated signal-notification register has a waiting unread signal value. Otherwise, the read-channel instruction will cause the SPU to stall until a write to the signal-notification register happens.

The following assembler-language program fragment shows how to read an SPU signal.

```
# Outputs:
#        $3 contains the signal control value read

rdch     $3, $SPU_RdSigNotify1
```

Here is a C-language program fragment that is comparable to the example above.

```
#include <spu_intrinsics.h>
unsigned int sn_value;

sn_value = spu_readch(SPU_RdSigNotify1); // Read SPU Signal Notification 1
```

19.7.7.2 *Waiting for Signals*

To avoid an SPU stall, SPE software can use the read-channel-count instruction on the signal channel of interest to determine if a signal has been sent. If the read-channel-count instruction returns ‘0’, no signal is pending. If the read-channel-count instruction returns a non-zero value, an unread signal is waiting, and the SPU can read the signal channel without stalling.

Poll For Signals

The following assembler-language program fragment shows how an SPU can poll one of its signal-notification channels.

```
# Outputs:
#         $3 contains the signal control word 1

repeat:
   rchcnt $3, $SPU_RdSigNotify1
   brz    $3, repeat
```

```
rdch      $3, $SPU_RdSigNotify1
```

Here is a C-language program fragment that is comparable to the example above.

```
#include <spu_intrinsics.h>
unsigned int sn_value;

do {
    /* Do other useful work while waiting. */
} while (!spu_readchcnt(SPU_RdSigNotify1));

sn_value = spu_readch(SPU_RdSigNotify1); // Read the SPU Signal Notification 1 Channel
```

Poll or Block for Signal-Notification Events

The SPU can also avoid a stall by enabling the SPU Event Facility, described in *Section 18* on page 463. Using events involves the following steps:

1. Enabling an event by setting the Signal Notification 1 Available event bit or the Signal Notification 2 Available event bit in the SPU Write Event Mask Channel.
2. Monitor for this event. Software can poll with a read-channel-count instruction on SPU Read Event Status; software can block (stall) with a read-channel instruction on the SPU Read Event Status Channel; software can enable SPU interrupts and use interrupt handlers to service signal-notification events as they are encountered asynchronously.
3. When the event is detected, it must be acknowledged by writing the corresponding bit to the SPU Write Event Acknowledgement channel. A typical handler for this event would perform the acknowledge and also read from the appropriate SPU Signal Notification channel.

The following assembler-language program fragment shows how to poll for or block on signal-notification events. To block instead of poll, simply delete the repeat loop.

```
# Outputs:
#   $3 contains the signal notification register contents
#   $4 contains the event that occurred

receive_signal:
    ila    $3, 0x200
    wrch $SPU_WrEventMask, $3# Write event mask

repeat:           # Wait for event to occur
    rchcnt $3, $SPU_RdEventStat
    brz    $3, repeat

    rdch   $4, $SPU_RdEventStat# Read the events that occurred
    wrch   $5, $SPU_WrEventStat, $4# Ack the events received
    rdch   $3, $SPU_RdSigNotify1# Read the signal notification register
```

Here is a C-language program fragment that is comparable to the example above. To block, instead of poll, simply delete the do/while loop below.

```
#include <spu_intrinsics.h>
#include <spu_mfcio.h>

unsigned int event_status;
unsigned int sn_value;

spu_writech(SPU_WrEventMask, MFC_SIGNAL_NOTIFY_1_EVENT);  // Enable event

do {
    /* Do other useful work while waiting. */
} while (!spu_readchcnt(SPU_RdEventStat));

event_status = spu_readch(SPU_RdEventStat);               // Read event status

spu_writech(SPU_WrEventAck, event_status);                // Ack. event

sn_value = spu_readch(SPU_RdSigNotify1);                   // Read the signal value
```

19.7.8 Differences Between Mailboxes and Signal Notification

For a comparison of the differences between the mailbox and signal-notification facilities, see *Table 19-2* on page 506.

20. Shared-Storage Synchronization

This chapter treats the subject of synchronization in two aspects. The first involves storage-access and instruction-execution ordering for shared-storage environments. The second involves the use of atomic operations to create semaphores, mutex locks, and other synchronization primitives for shared-storage environments.

Synchronization should be done with care because it affects performance. For example, one key to successful parallelization strategies is the minimizing of synchronizing events between the computational elements.

This chapter describes only non-privileged (user) synchronization facilities and scenarios. Some operations on privileged registers and memory-management facilities also require synchronization. For details, see *Section 4 Virtual Storage Environment* on page 79, *Section 9 PPE Interrupts* on page 237, *Section 11 Logical Partitions and a Hypervisor* on page 327, and *PowerPC Operating Environment Architecture, Book III*.

20.1 Shared-Storage Ordering

This section describes storage-access and instruction-execution ordering for shared-storage environments. The CBE processor contains multiple shared-storage domains—the main-storage domain, eight local LS-address domains, and eight local channel domains—as illustrated in *Figure 1-2* on page 40.

20.1.1 Storage Model

Although the CBE processor executes instructions in program order, it loads and stores data using a *weakly consistent* storage model. That is, the order in which any processor element (PPE or SPE) performs storage accesses, the order in which those accesses are performed with respect to another processor element or mechanism, and the order in which those accesses are performed in main storage may be different. This storage model allows storage accesses to be reordered dynamically, which provides an opportunity for improved overall performance and reduced effect of memory latency on instruction throughput. Therefore, in a memory environment that is shared by other threads of execution or I/O devices, this weakly consistent model requires that programs explicitly order accesses to storage if they want stores to occur in the program order.

To ensure that accesses to shared storage (main storage, which includes external access to the LSs of SPEs) are performed in program order, software must place memory-barrier instructions between storage accesses, as required by the *Cell Broadband Engine Architecture* (CBEA). Programs that do not contain such memory-barrier instructions in appropriate places might be executed correctly for a given implementation or execution environment but will fail if the execution environment is changed or is executed on another implementation of the CBEA.

The term *storage access* means an access to main storage caused by a load, a store, a DMA read, or a DMA write. There are two orders to consider:

- *Order in which Instructions are Executed*—Instructions can be executed in-order or out-of-order. For an in-order machine, such as the CBE processor, each instruction completes before the next instruction starts. In general, from the viewpoint of a program, it appears that the instructions are executed in the order specified by the program. An out-of-order machine

does not guarantee that the processor element executes instructions in program order. A program cannot detect that a sequence of instructions is executed in an order different than specified in the program.

- *Order in which Storage Accesses are Performed*—The order in which shared-storage accesses are performed may be different from both program order and the order in which the instructions that caused the accesses are executed. Consider a program that contains a sequence of loads from locations A, B, and C, in that order. The processor element might execute these instructions in a different order (for example, B, A, C) and perform the accesses caused by the instructions in yet a different order (for example, C, B, A).

The CBEA defines four storage-control attributes that are maintained by the operating system in the page-table entries (see *Section 4.2.6 Paging* on page 87):

- *Write-Through Required (W attribute)*—When an access is designated as write-through, if data is in a cache (such as one of the PPE's L1 or L2 caches), a store operation updates both the cache copy of the data and the main-storage location of the data.
- *Caching-Inhibited (I attribute)*—When an access is designated as caching-inhibited, the access is completed by referencing only the location in main storage, bypassing the cache.
- *Memory-Coherence Required (M attribute)*—When an access is designated as memory-coherent, a hardware indication is sent to the rest of the system indicating that the access is global. Other processor elements affected by the access must acknowledge and respond appropriately to the access.
- *Guarded (G attribute)*—Memory is marked as guarded when areas of main storage must be protected from accesses not directly dictated by the program. For example, the guarded attribute can be used to prevent out-of-order, speculative load, or prefetch operations from occurring with peripheral devices; such accesses can produce undesirable results.

Main storage has the memory-coherence required attribute. SPE LSs and I/O devices that are mapped to the effective-address space usually have the cache-inhibited and guarded attributes.

Several means of enforcing an ordering of storage accesses are provided to allow programs to share storage with other programs or threads of execution, or with other mechanisms such as I/O devices.The appropriate mechanism depends upon the subsystem wishing to enforce storage ordering and the storage-control attributes of the storage being accessed.

Table 20-1 summarizes the effects on address and communication domains of the synchronization instructions, commands, and facilities that are described in the sections immediately following this table. Gray shading in a table cell means that the instruction, command, or facility has no effect on the referenced domain.

Table 20-1. Effects of Synchronization on Address and Communication Domains[1]

Issuer	Instruction, Command, or Facility	**Main-Storage Domain**			**LS Domain[2]**			**Channel Domain[3]**
		Accesses by PPE		Accesses by All Other Processor Elements and Devices	Accesses by Issuing SPU	Accesses by Issuing SPU's MFC	Accesses by All Other Processor Elements and Devices	Accesses by Issuing SPU
		Issuing Thread	Both Threads					
PPU	**sync[4]**	all accesses				Unreliable. Use MFC Multisource Synchronization Facility[5]		
	lwsync[6]	accesses to memory-coherence-required locations						
	eieio	accesses to caching-inhibited and guarded locations		accesses to caching-inhibited and guarded locations		Unreliable. Use MFC Multisource Synchronization Facility[5]		
	isync	instruction fetches						
SPU	**sync**				all accesses			all accesses
	dsync				load and store accesses			
	syncc				all accesses			
MFC	**mfcsync**			all accesses		Unreliable. Use MFC Multisource Synchronization Facility[5]		
	mfceieio			accesses to caching-inhibited and guarded locations				
	barrier			all accesses				
	<f>, <b>			all accesses for the tag group				
	MFC Multisource Synchronization Facility			all accesses	all accesses			

1. Gray shading in a table cell means that the instruction, command, or facility has no effect on the referenced domain.
2. The LS of the issuing SPE.
3. The channels of the issuing SPE.
4. This is the PowerPC **sync** instruction with L = '0'.
5. These accesses can exist only if the LS is mapped by the PPE operating system to the main-storage space. This can only be done if the LS is assigned caching-inhibited and guarded attributes.
6. This is the PowerPC **sync** instruction with L = '1'.

20.1.2 PPE Ordering Instructions

The PPE supports *barrier instructions* for ordering storage accesses and instruction execution. The *PowerPC Architecture* defines two types of barrier instructions:

- Storage Barriers—The **sync**, **lwsync**, **ptesync**, and **eieio** instructions.
- Instruction Barrier—The **isync** instruction.

These instructions can be used between storage-access instructions to define a memory barrier that divides the instructions into those that precede the barrier instruction and those that follow it. The storage accesses caused by instructions that precede the barrier instruction are performed before the storage accesses caused by instructions that follow the barrier instruction.

The PPE supports one additional synchronization instruction, **tlbsync**, that is privileged. For details, see *Section 4 Virtual Storage Environment* on page 79.

20.1.2.1 *The PPE sync Instruction*

The PPE **sync** instruction—which differs from the SPU **sync** instruction (*Section 20.1.3.1* on page 559)—ensures that all instructions preceding the **sync** appear to have completed before the **sync** instruction completes, and that no subsequent instructions are initiated until after the **sync** instruction completes. This does not mean that the previous storage accesses have completed before the **sync** instruction completes.

The **sync** instruction is also known as the *heavyweight* **sync** instruction, as opposed to the light-weight **lwsync** instruction and the **ptesync** instruction. The opcode of the **sync** instruction has its L bit cleared to ‘0’, whereas the **lwsync** instruction is the **sync** instruction with L = ‘1’ and the the **ptesync** instruction is the **sync** instruction with L = ‘2’. See the *PowerPC Operating Environment Architecture, Book III* for details.

The **sync** instruction orders all main-storage accesses, regardless of the main-storage memory attributes. For example, it orders a caching-inhibited load with respect to a caching-enabled store. However, only processor elements operating in the same logical partition (see *Section 4 Virtual Storage Environment* on page 79) will accept the **sync**; processor elements in other logical partitions may ignore it.

In addition, the **sync** instruction does the following:

- Prevents store-combining to storage that is caching-inhibited or write-through-required
- Prevents load-combining for storage that is caching-inhibited.
- Is cumulative—that is, all processor elements in the system will see the same barrier sequence that is performed and seen by the processor element that executes the **sync** instruction (the PPE).

The **sync** instruction can be used, for example, to ensure that the results of all stores into a data structure, caused by store instructions executed in a critical section of a program, are seen by other processor elements before the data structure is seen as unlocked.

The functions performed by the **sync** instruction normally take a significant amount of time to complete, and the time required to execute **sync** may vary from one execution to another, so this instruction should be used with care. The **eieio** instruction may be more appropriate than **sync** for many cases (see *Table 20-3* on page 557 and *Table 20-4* on page 558).

20.1.2.2 *The lwsync Instruction*

The **lwsync** instruction (also known as the *lightweight* **sync** instruction) creates the same barrier as the **sync** instruction for storage accesses that have the memory-coherence-required attribute. However, the **lwsync** instruction does not create a barrier for storage accesses that have either the caching-inhibited or write-through-required attribute.

Unlike the **sync** instruction, the **lwsync** instruction orders only the main-storage accesses of the PPE. It has no effect on the main-storage accesses of other processor elements in the system. Thus, the effects of the **lwsync** instruction are not cumulative.

The **lwsync** instruction should be used when ordering is required only for coherent memory, because **lwsync** executes faster than **sync**.

20.1.2.3 *The eieio Instruction*

The **eieio** instruction (enforce in-order execution of I/O) ensures that all main-storage accesses caused by instructions proceeding the **eieio** have completed, with respect to main storage, before any main-storage accesses caused by instructions following the **eieio**.

The **eieio** instruction orders loads and stores in the following two sets, which are ordered separately in the following sequence:

1. Loads and stores to storage that is both caching-inhibited and guarded, and stores to storage that is write-through-required. All accesses in this set are ordered as a single set—that is, there is not one order for loads and stores to caching-inhibited and guarded memory and another order for stores to write-through-required memory.
2. Stores to storage that is memory-coherence-required but that is neither caching-inhibited nor write-through-required.

In addition, the **eieio** instruction does the following:

- Prevents store-combining to storage that is both caching-inhibited and guarded, or to storage that is write-through-required.
- Prevents load-combining for storage that is caching-inhibited.
- Is cumulative (see the **sync** instruction, above) for storage that is neither caching-inhibited nor write-through-required.

The *PowerPC Architecture* specifies that stores to storage that is both caching-inhibited and guarded are performed in program order, so **eieio** is needed for such storage only when loads must be ordered with respect to stores or with respect to other loads, or when load and store combining operations must be prevented.

An **eieio** instruction issued on one PPE thread has no architected effect on the other PPE thread. However, there is a performance effect because all load or store operations (cacheable or non-cacheable) on the other thread are serialized behind the **eieio** instruction.

The **eieio** instruction does not order accesses with differing storage attributes. For example, if an **eieio** is placed between a caching-enabled store and a caching-inhibited store, the access could still be performed in an order different than specified by the program.

The **eieio** instruction is intended for use in managing shared data structures, in accessing memory-mapped I/O, and in preventing load or store combining. For the first use, the shared data structure and the lock that protects it must be altered only by stores that are in the same **eieio** ordering set (see above). For the second use, **eieio** can be thought of as placing a barrier into the stream of memory accesses issued by a processor element, such that any given memory access appears to be on the same side of the barrier to both the processor element and the I/O device.

20.1.2.4 *The isync Instruction*

The **isync** instruction ensures that all PPE instructions proceeding the **isync** are completed before **isync** is completed. An **isync** (L = 0 or 1) instruction causes issue stall and blocks all other instructions from both PPE threads until the **isync** instruction completes.

Prefetched instructions on the issuing thread are discarded; those on the other PPE thread are unaffected. The **isync** instruction only flushes based on thread. The instructions from the other thread which are already fetched or dispatched are not refetched after the **isync** completes.

This instruction is most often used in conjunction with self-modifying PPE code. After an instruction is modified, an **isync** instruction must be issued prior to execution of the modified instruction. The **isync** instruction may also be used during context-switching when the MMU's translation rules are being changed.

The **isync** instruction is *context synchronizing*, as opposed to the **sync** instruction which is *execution synchronizing*. See the *Glossary* on page 823 and the Synchronization section of *PowerPC Operating Environment Architecture, Book III* for a further information.

20.1.2.5 *Summary of Storage-Barrier Instruction Use*

Table 20-2 summarizes the functions of the storage-barrier instructions with respect to storage attributes. In this table, "yes" (and "no") mean that the instruction performs (or does not perform) a barrier function on storage with the related attribute.

Table 20-2. Storage-Barrier Function Summary

Storage Attribute or Function	Loads and/or Stores	**sync**	**lwsync**	**eieio**
memory-coherence-required	loads	yes	yes	no
	stores			yes, if neither caching-inhibited nor write-through-required[1]
caching-inhibited	loads and stores		no	yes, if both caching-inhibited and guarded[2]
guarded	loads and stores			
write-through-required	stores			yes[2]
	loads	not applicable		
combining prevented	loads	for caching-inhibited		
	stores	for (a) caching-inhibited, and for (b) write-through-required		for (a) caching-inhibited and guarded, and for (b) write-through-required
cumulative[3]	loads and stores	yes	no	yes, if neither caching-inhibited nor write-through-required

1. Second ordering: stores to storage that is memory-coherence-required but that is neither caching-inhibited nor write-through-required.
2. First ordering: loads and stores to storage that is both caching-inhibited and guarded, and stores to storage that is write-through-required
3. All processor elements in the system will see the same barrier sequence that is performed and seen by the processor element that executes the instruction.

Table 20-3 and *Table 20-4* summarize the use of the storage-barrier instructions for two common types of main-storage memory:

- *System Memory*—Memory that has the memory-coherence-required attribute.
- *Device Memory*—Memory that has caching-inhibited and guarded attributes, which is typical of memory-mapped I/O devices. (The mapping of SPEs' LS to main storage is generally caching-inhibited but not guarded.)

In these tables, "yes" (and "no") mean that the instruction performs (or does not perform) a barrier function on the related storage sequence, "recommended" means that the instruction is the preferred one, "not recommended" means that the instruction will work but is not the preferred one, and "not required" and "no effect" mean the instruction has no effect.

Table 20-3. Storage-Barrier Ordering of Accesses to System Memory

Storage-Access Instruction Sequence	**sync**	**lwsync**	**eieio**
load-*barrier*-load	yes	recommended	no affect
load-*barrier*-store	yes	recommended	no affect
store-*barrier*-load	yes	no	no affect
store-*barrier*-store	yes	recommended	not recommended

Table 20-4. Storage-Barrier Ordering of Accesses to Device Memory

Storage-Access Instruction Sequence	**sync**	**lwsync**	**eieio**
load-*barrier*-load	yes	no affect	yes
load-*barrier*-store	yes	no affect	yes
store-*barrier*-load	yes	no affect	yes
store-*barrier*-store	not required [1]	no affect	not required [1]

1. Two stores to caching-inhibited storage are performed in the order specified by the program, regardless of whether they are separated by a barrier instruction or not.

The following rules apply to ordering operations for cacheable and caching-inhibited locations:

- The ordering of cacheable loads is not guaranteed. If ordering is required, software needs to issue a **sync** or **lwsync** instruction.
- Caching-inhibited loads are implemented as guarded operations. This means that the PPE does not issue a new load until the data for the older load is returned. There can only be one outstanding caching-inhibited load at any time.
- Caching-inhibited instruction fetches are treated as nonguarded operations. The PPE does not maintain ordering for caching-inhibited, nonguarded operations.

20.1.3 SPU Ordering Instructions

An LS can experience asynchronous interaction from the following streams that access it:

- Instruction fetches by the local SPU.
- Data loads and stores by the local SPU.
- DMA transfers by the local MFC or another SPE's MFC.
- Loads and stores in the main-storage space by other processor elements or devices.

With regard to an SPU, the CBE's in-order execution model[48] guarantees only that SPU instructions that access that SPU's LS appear to be performed in program order with respect to that SPU. These accesses might not appear to be performed in program order with respect to external accesses to that LS or with respect to the SPU's instruction fetch. Thus, in the absence of external LS writes, an SPU load from an address in its LS returns the data written by that SPU's most-recent store to that address. However, an instruction fetch from that address does not necessarily return that data. The SPU may buffer and otherwise reorder its LS accesses. Instruction fetches, loads, and stores can access the LS in any order. The SPU can speculatively read the LS. However, the SPU does not speculatively write the LS; the SPU only writes the LS on behalf of stores required by the program.

The SPU ISA provides three synchronization instructions: synchronize (**sync**), synchronize data (**dsync**), and synchronize channel (**syncc**). These instructions have both coherency and instruction-serializing effects.

In the descriptions that follow, the following terms are used:

48. The CBE's in-order execution model is implementation-specific, rather than part of the *Cell Broadband Engine Architecture*.

- “External Read” and “External Write” mean accesses by any DMA transfer (by the local MFC or the MFC associated with another SPE) or any processor element (including the PPE) or other device—other than the SPU that executes the synchronization instruction.
- “SPU Load” and “SPU Store” mean accesses by the SPU that executes the synchronization instruction.

20.1.3.1 *The SPU sync Instruction*

The SPU **sync** instruction—which differs from the PPE **sync** instruction (*Section 20.1.2.1* on page 554)—causes the SPU to wait until all pending store instructions to its LS have completed before fetching the next sequential instruction. It orders instruction fetches, loads, stores, and channel accesses.

After an SPU has executed a **sync** instruction, the SPU’s instruction fetches from an LS address return data stored by the most-recent store instruction or external write to that address. In addition, the SPU will have completed all prior loads, stores, and channel accesses and will not have begun execution of any subsequent loads, stores, or channel accesses. At this time, an external read from an LS address returns the data stored by the most-recent SPU store to that address. SPU loads after the **sync** return the data externally written prior to the moment when the **sync** completes.

The **sync** instruction affects only that SPU’s instruction sequence and the coherency of that SPU’s fetches, loads, and stores, with respect to actual LS state. The SPU does not broadcast **sync** notification to external devices that access its LS, and therefore the **sync** does not affect the state of external devices.

The SPU’s instruction-fetch buffers and pipelines are flushed when it executes the **sync** instruction. The **sync** instruction has no affect on, and does not wait for, DMA **get** operations controlled by the SPU’s MFC.

The **sync** instruction must be used before attempting to execute new code that either arrives through DMA transfers or is written with store instructions. This instruction is most often used in conjunction with self-modifying SPE code.

The CBE processor invalidates the instruction prefetch buffer whenever a mispredicted branch is executed. In such a case, the **sync** instruction can be omitted, but for architecture portability the **sync** instruction should still be used.

20.1.3.2 *The dsync Instruction*

The **dsync** (data synchronization) instruction allows SPE software to ensure that data has been stored in the LS before the data becomes visible to the local MFC or other external devices. It orders loads, stores, and channel accesses but not instruction fetches, and it does not affect any prefetching of instructions that the SPE may have done.

After a **dsync** completes, the SPU will have completed all prior loads, stores, and channel accesses and will not have begun execution of any subsequent loads, stores, or channel accesses. At this time, an external read from an LS address returns the data stored by the most-recent SPU store to that address. SPU loads after the **dsync** return the data written prior to the moment when the **dsync** completes.

The **dsync** instruction affects only that SPU's instruction sequencing and the coherency of that SPU's loads and stores, with respect to actual LS state. The SPU does not broadcast **dsync** notification to external devices that access its LS, and therefore the **dsync** does not affect the state of the external devices.

DMA transfers can interfere with store instructions and the store buffer. Architecturally, the **dsync** instruction is used to ensure that all store buffers have been flushed to the LS, so that all previous stores to the LS will be seen by subsequent LS accesses. However, the initial implementation of the CBE processor does not require the **dsync** instruction for this purpose.

20.1.3.3 *The syncc Instruction*

The **syncc** (channel synchronization) instruction performs channel synchronization followed by the same synchronization provided by the **sync** instruction. It forces all channel instructions to complete before executing the next SPU instruction. It ensures that the effects on SPU state caused by prior **wrch** instructions are propagated and influence the execution of the following instructions.

Data through a particular channel is never reordered. A channel is either blocking or non-blocking. If a channel is blocking, read data must be present or previous write data must have been consumed before the instruction that accesses the channel can complete. Thus, the hardware synchronizes the channel streams. However, if a channel is non-blocking, then software must manage the synchronization based on the ordering of channel data.

Some channel activity may, as a side effect, alter the SPU operating state. These side effects are not ordered with respect to the SPU pipeline. Such side effects can be synchronized with SPU-instruction execution using the **syncc** instruction. Channel synchronization does not ensure that DMA commands generated as a result of issuing a channel command have completed; it ensures only that such DMA commands have been placed in the MFC command queue.

See *Section 17* on page 441 for details about channels.

20.1.3.4 *Summary of SPU Ordering Instructions*

The SPU synchronization instructions are summarized in *Table 20-5* on page 561. *Table 20-6* shows which SPU synchronization instructions are required between LS writes and LS reads to ensure that reads access data written by prior writes.

Table 20-5. SPU Synchronization Instructions

Instruction	Coherency Effects	Instruction Serialization Effects
sync	Ensures that subsequent external[1] reads access data written by prior SPU stores. Ensures that subsequent instruction fetches access data written by prior SPU stores and external[1] writes. Ensures that subsequent SPU loads access data written by external[1] writes.	Forces all access of LS and channels due to instructions prior to the **sync** to be completed prior to completion of **sync**. Forces all access of LS and channels due to instructions after the **sync** to occur after completion of the **sync**.
dsync	Ensures that subsequent external[1] reads access data written by prior SPU stores. Ensures that subsequent SPU loads access data written by external[1] writes.	Forces SPU load and SPU store access of LS due to instructions prior to the **dsync** to be completed prior to completion of **dsync**. Forces read-channel operations due to instructions prior to the **dsync** to be completed prior to completion of the **dsync**. Forces SPU load and SPU store access of LS due to instructions after the **dsync** to occur after completion of the **dsync**. Forces read- and write-channel operations due to instructions after the **dsync** to occur after completion of the **dsync**.
syncc	Ensures that subsequent external[1] reads access data written by prior SPU stores. Ensures that subsequent instruction fetches access data written by prior SPU stores and external[1] writes. Ensures that subsequent SPU loads access data written by external[1] writes. Ensures that subsequent instruction processing is influenced by all internal execution states modified by previous **wrch** instructions.	Forces all access of LS and channels due to instructions prior to the **syncc** to be completed prior to completion of **syncc**. Forces all access of LS and channels due to instructions after the **syncc** to occur after completion of the **syncc**.

1. By any DMA transfer (from the local MFC or a non-local MFC), the PPE, or other device—other than the SPU that executes the synchronization instruction.

Table 20-6. Synchronization Instructions for Accesses to an LS

Writer	Reader		
	SPU Instruction Fetch	SPU Load	External[1] Read
SPU Store	**sync**	nothing required	**dsync**
External[1] Write	**sync**	**dsync**	not applicable

1. By any DMA transfer (from the local MFC or a non-local MFC), the PPE, or other device—other than the SPU that executes the synchronization instruction.

20.1.3.5 *Ordering and Synchronization of External Accesses to the MFC and LS*

Privileged software on the PPE can make the SPE's LS and MFC resources accessible to the PPE or other devices in the main-storage space. If two accesses are made to two different addresses, no ordering is maintained between the accesses unless the PPE **eieio** or **sync** instruction is used between the two accesses. (The **eieio** instruction can be used because LS mappings to main storage have the caching-inhibited attribute.)

20.1.4 MFC Ordering Mechanisms

The MFC of each SPE supports synchronization commands and command options that control the order in which DMA storage accesses are performed by that MFC. These synchronization commands are of two kinds:

- *Barrier Commands*—The **mfcsync**, **mfceieio**, and **barrier** commands order storage accesses made through the MFC with respect to all other MFCs, processor elements, and other devices in the system. The CBEA specifies that the **mfcsync** and **mfceieio** commands are tag-specific, but the first implementation of the *CBEA* (the CBE processor) treats all three barrier command identically, having no tag-specific effects.
- *Fence or Barrier Command Options*—These are options of the **get**, **put**, and **sndsig** commands. The options order storage accesses and signaling only for a specific, local tag group and MFC command queue.

For an overview of MFC commands, command queues, and tag groups, see *Section 19.2* on page 506.

20.1.4.1 *The mfcsync Command*

The **mfcsync** command is similar in operation to the PPE **sync** instruction, described in *Section 20.1.2.1* on page 554. The CBEA specifies that the **mfcsync** command controls the order in which MFC commands within a specified tag group for that MFC are executed with respect to storage accesses by all other processor elements and devices in the system. The **mfcsync** command and all subsequent commands in the same tag group are ordered with respect to all previous commands in the queue with the same tag ID.

Although the CBEA specifies the **mfcsync** command as having tag-specific effects, the first implementation of the *CBEA* (the CBE processor) treats this command as not having tag-specific effects and as behaving the same as the **barrier** command, described in *Section 20.1.4.3* on page 563.

The C-intrinsic syntax for an SPE issuing an **mfcsync** command with the `tag_id` tag group is:

```
spu_writech(MFC_TagID, tag_id);
spu_writech(MFC_Cmd, 0xCC);
```

To obtain the best performance across the widest range of implementations, use the **mfcsync** command only when its entire functionality is needed. Use the **mfceieio** command, the **barrier** command, or fence **<f>** or barrier **<b>** options of other commands, if any of these are sufficient for the ordering needs.

If the storage accesses between two DMA operations that access storage having different storage attributes need to be strongly ordered, the fence **<f>** or barrier **<b>** options of the **get** and **put** commands can do this. However, these options are insufficient to guarantee the ordering function with respect to other processor elements or devices. In these cases, an **mfcsync** command must be issued between commands involving storage with different storage attributes to provide the required ordering function.

20.1.4.2 *The mfceieio Command*

The **mfceieio** command is similar in operation to the PPE **eieio** instruction, described in *Section 20.1.2.3* on page 555. The CBEA specifies that the **mfceieio** command controls the order in which DMA commands within a specified tag group for that MFC are executed with respect to storage accesses by all other processor elements and devices in the system, when the storage being accessed has the attributes of caching-inhibited and guarded (typical for I/O devices).

Although the CBEA specifies the **mfceieio** command as having tag-specific effects, the first implementation of the *CBEA* (the CBE processor) treats this command as not having tag-specific effects and as behaving the same as the **barrier** command, described in *Section 20.1.4.3* on page 563.

The **mfceieio** command only orders:

- **get** commands with respect to other **get** commands from storage that is caching-inhibited and guarded.
- **get** commands with respect to **put** commands accessing storage that is caching-inhibited and guarded.
- **put** commands with respect to **put** commands accessing storage that is memory-coherence-required and is not caching-inhibited.

The C-intrinsic syntax for issuing an **mfceieio** command with the `tag_id` tag group is:

```
spu_writech(MFC_TagID, tag_id);
spu_writech(MFC_Cmd, 0xC8);
```

The **mfceieio** command is intended for use in managing shared data structures, in performing memory-mapped I/O, and in preventing load and store combining in main storage. In order to obtain the best performance across the widest range of implementations, use the fence **<f>** or barrier **<b>** options of other commands, if any of these are sufficient for the ordering needs.

20.1.4.3 *The barrier Command*

The **barrier** command orders all subsequent MFC commands with respect to all MFC commands preceding the **barrier** command in the DMA command queue, independent of tag groups. The **barrier** command will not complete until all preceding commands in the queue have completed. After the command completes, subsequent commands in the queue may be started.

The C syntax for issuing a **barrier** command is:

```
spu_writech(MFC_TagID, tag_id);
spu_writech(MFC_Cmd, 0xC0);
```

The setting of the tag ID is optional and need only be provided to assign an ID to the **barrier** command for subsequent tag-completion testing.

In contrast to the **barrier** command, which operates independently of tag-groups, the fence **<f>** or barrier **<b>** options of **get** and **put** commands form of fence or barrier that only affects commands within the same tag group.

20.1.4.4 *Tag-Specific Ordering Commands*

Local ordering of the MFC commands within a specific tag group can be achieved using the fence **<f>** or barrier **<b>** options of the **get**, **put**, or **sndsig** commands. Local ordering only ensures ordering of the MFC commands with respect to that particular MFC command queue and tag group (tag-groups cannot span multiple MFCs).

The storage system may complete requests in an order different then the order in which they are issued, depending on the storage attributes. However, caching-inhibited storage access are ordered in the order in which they are issued. *Table 20-7* lists the tag-specific ordering commands.

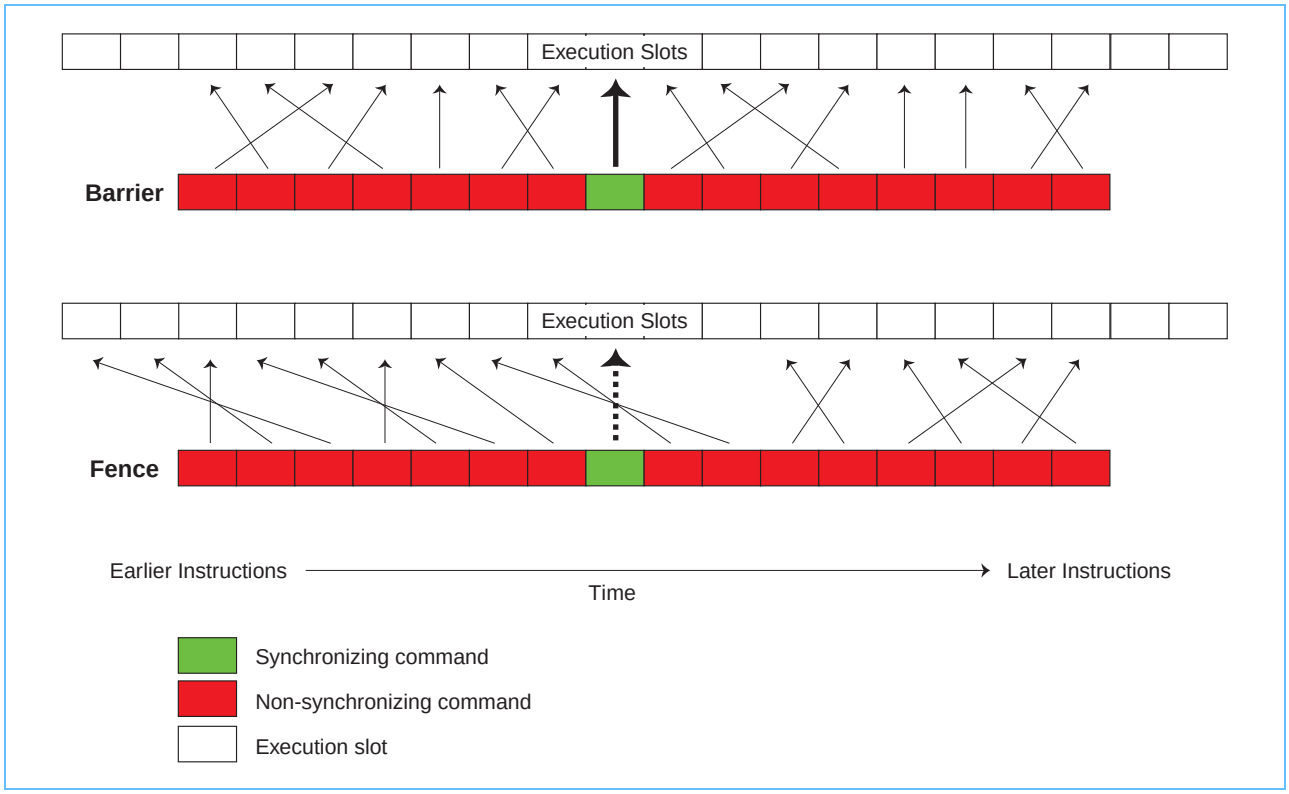
Table 20-7. Tag-Specific Ordering Commands

Option	Commands
barrier	**getb, getbs, getlb, putb, putbs, putrb, putlb, putrlb, sndsigb**
fence	**getf, getfs, getlf, putf, putfs, putrf, putlf, putrlf, sndsigf**

The fence form ensures that all previous MFC commands of the same tag group are issued prior issuing the fenced command. The barrier form ensures that all previous MFC commands in the same tag group are issued prior to issuing the barrier command, and that none of the subsequent MFC commands in the same tag group are issued prior to the barrier command.

Figure 20-1 on page 565 illustrates the behavior of barriers and fences. In this example, the row of white boxes represents command-execution slots, in real-time, in which the DMA commands (red and green boxes) might execute. Each DMA command is assumed to transfer the same amount of data, thus, all boxes are the same size. The arrows show how the DMA hardware, using out-of-order execution, might execute the DMA commands over time. So, a barrier does not allow the DMA hardware to reach across the barrier to find eligible commands to execute. A fence, in contrast, allows the DMA hardware to reach across the fence in one direction to find eligible DMA commands to run.

Figure 20-1. Barriers and Fences



Storage-Access Consistency for DMA-Command Combinations

The fence and barrier options provide stronger storage-access consistency for some combinations of DMA commands:

- A **put(l)** command followed by a **put(l)** command with either the fence **<f>** or barrier **<b>** option, where the storage attributes are memory-coherence-required and neither write-through-required nor caching-inhibited, has a storage-order effect at least as strong as two stores on the PPE separated by a **lwsync** instruction.
- A **get(l)** command followed by a **get(l)** command with either the fence **<f>** or barrier **<b>** option, where the storage attributes are memory-coherence-required and neither write-through-required nor caching-inhibited, has a storage-order effect at least as strong as two loads on the PPE separated by a **lwsync** instruction.
- A **put(l)** command followed by a **put(l)** command with either the fence **<f>** or barrier **<b>** option, where the storage attribute is caching-inhibited, has a storage-order effect at least as strong as two stores on the PPE separated by a **sync** instruction.
- A **get(l)** command followed by a **get(l)** command with either the fence **<f>** or barrier **<b>** option, where the storage attributes are caching-inhibited and not guarded, has a storage-order effect at least as strong as two loads on the PPE separated by a **sync** instruction.
- A **get(l)** command followed by a **get(l)** command with either the fence **<f>** or barrier **<b>** option, where the storage attributes are caching-inhibited and guarded, has a storage-order effect at least as strong as two loads on the PPE separated by an **eieio** instruction.

- A **put(l)** command followed by a **get(l)** command with either the fence **<f>** or barrier **<b>** option, where the storage attributes are caching-inhibited and not guarded, has a storage-order effect at least as strong as a store followed by a load on the PPE separated by a **sync** instruction.
- A **get(l)** command followed by a **put(l)** command with either the fence **<f>** or barrier **<b>** option, where the storage attributes are memory-coherence-required and neither write-through-required nor caching-inhibited, has a storage-order effect at least as strong as a load followed by a store on the PPE separated by a **lwsync** instruction.
- A **get(l)** command followed by a **put(l)** command with either the fence **<f>** or barrier **<b>** option, where the storage attributes are caching-inhibited and not guarded, has a storage-order effect at least as strong as a load followed by a store on the PPE separated by a **sync** instruction.
- A **get(l)** command followed by a **put(l)** command with either the fence **<f>** or barrier **<b>** option, where the storage attributes are caching-inhibited and guarded, has a storage-order effect at least as strong as a load followed by a store on the PPE separated by an **eieio** instruction.
- A **put(l)** command followed by a **put(l)** command with either the fence **<f>** or barrier **<b>** option, where the storage attributes are caching-inhibited and guarded, are always performed in program order with respect to any processor element or mechanism.

Common Fenced-Option Use

The most common use of the fenced DMA is when writing back notifications. Consider an example in which an SPE computes some data, writes the computed data back to main storage, then writes a notification that the data is available. This notification could be to any type memory (main memory, I/O memory, an MMIO register, a signal-notification register, or another SPE's mailbox). To ensure ordering of the DMA write and the notification, the notification can be sent using a fenced DMA command, so that the notification is not sent until all previous DMA commands of the group are issued.

A fence option may also be useful when multiple DMA commands are needed to load an SPU program and to start its execution. In this example, one DMA command is used to load the initialized data segment and a second DMA command with both the SPU start (''**s**") suffix (causing execution to start after the DMA transfer completes) and the tag-specific fence ("**f**") suffix is used to load the code (the "text" segment). If the two commands have the same tag ID, the fence ensures that the load of the data segment is completed before loading the text data and before starting the SPU program execution. Without the fence, the second DMA command could complete and could start the SPU program before the data segment is loaded.

Common Barrier-Option Use

A barrier option may be useful when a buffer read takes multiple commands and must be performed before writing the buffer, which also takes multiple commands. In this example, the commands to read the buffer can be queued and performed in any order. Using the barrier-form for the first command to write the buffer allows the commands used to write the buffer to be queued without waiting for an MFC tag-group event on the read commands. (The barrier-form is only required for the first buffer-write command.) If the buffer read and buffer write commands have the same tag ID, the barrier ensures that the buffer is not written before being read. If

multiple commands are used to read and write the buffer, using the barrier option allows the read commands to be performed in any order and the write commands to be performed in any order, which provides better performance but forces all reads to finish before the writes start.

Barrier commands are also useful when performing double-buffered DMA transfers in which the data buffers used for the input data is the same as the output data buffers. Consider the following example for such a double-buffered environment:

```
int i;
i = 0;
read buffer 0
while (more buffers) {

    read buffer i^1
    wait for buffer i
    compute buffer i
    write buffer i
    i = i^1;
}
wait buffer i
compute buffer i
write buffer i
```

At the bottom of the loop, data is written from the same buffer that is read into immediately at the top of the next loop iteration. It is critical that the writes complete before the reads are started. Therefore, the first read (at the top of the loop) should be a barrier read. (For examples of using double-buffering techniques to overlap DMA transfers with computation on an SPU, see *Section 24.1.2* on page 682.)

20.1.5 MFC Multisource Synchronization Facility

The PPE **sync** instruction (*Section 20.1.2.1* on page 554) provides cumulative ordering, such that all threads in the system see the same barrier sequence that is performed and seen by the PPE, one of whose two threads executes the **sync** instruction. However, **sync** is only cumulative with respect to the main-storage domain. The CBE processor contains multiple address and communication domains—the main-storage domain, eight local LS-address domains, and eight local channel domains—as illustrated in *Figure 1-2* on page 40. To ensure cumulative ordering across all address domains, the MFC Multisource Synchronization Facility must be used.

Standard PowerPC storage-ordering rules apply to storage accesses performed by one processor element or device with respect to another processor element or device. Ordering of storage accesses performed by multiple sources (that is, two or more processor elements or devices) with respect to another processor element or device is referred to as cumulative ordering, as described in the *PowerPC Virtual Environment Architecture, Book II*. Cumulative ordering is ensured when all accesses are performed in the main-storage (effective-address) domain and the proper synchronization instructions are performed. Cumulative ordering cannot be ensured when accesses are performed from both the main-storage and the LS-address domains.

The MFC Multisource Synchronization Facility addresses this cumulative-ordering need by providing two independent multisource synchronization-request methods:

- The MFC Multisource Synchronization Register, which allows the PPE or other processor elements or devices to control synchronization from the main-storage domain.
- The MFC Write Multisource Synchronization Request Channel, which allows an SPE to control synchronization from its LS-address and channel domain.

Each of these two synchronization-request methods ensures that all write transfers to the associated MFC are sent and received by the MFC before the MFC synchronization-request is completed. This facility does not ensure that read data is visible at the destination when the associated MFC is the source.

The channel and register supporting the MFC Multisource Synchronization Facility are shown in *Table 20-8*. For details about channels and their associates MMIO registers, see *Section 17* on page 441.

Table 20-8. MFC Multisource Synchronization Facility Channel and MMIO Register

SPE Channel #	Name	Channel Interface					MMIO Register Interface				
		Mnemonic	Max. Entries	Blocking	R/W	Width (bits)	Offset From Base	Mnemonic	Max. Entries	R/W	Width (bits)
9	MFC Write Multisource Synchronization Request	MFC_WrMSSyncReq	1	yes	W	32	x'00000'	MFC_MSSync	1	R/W	64

These two synchronization-request methods operate independently. A synchronization request through the MMIO register has no effect on synchronization requests through the channel, and vice versa.

All outstanding requests that are in an SPE's snoop-write queue when a Multisource Synchronization Request is received are completed before reporting the completion of the Multisource Synchronization Request.

20.1.5.1 *MFC Multisource Synchronization Register*

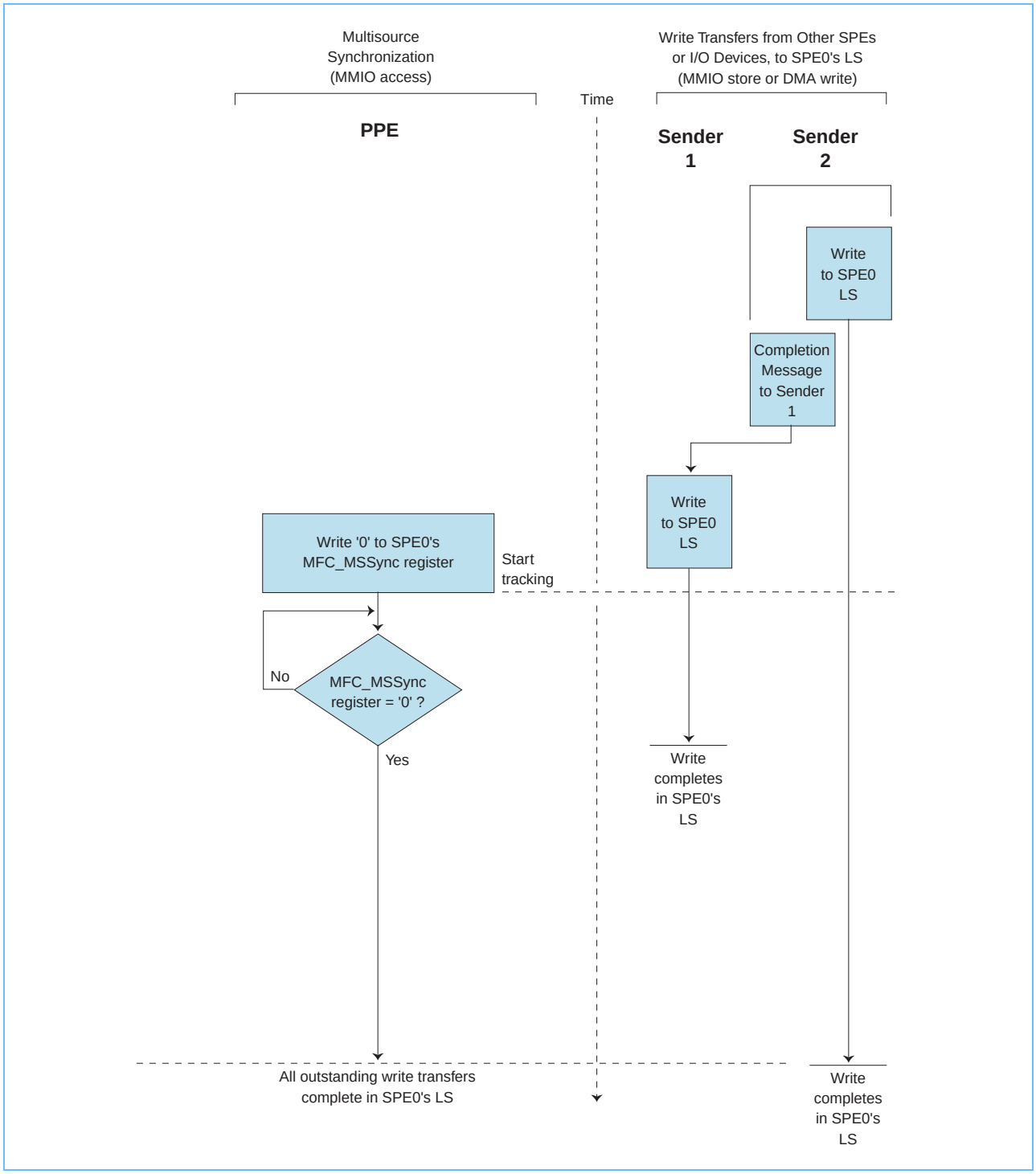
The MFC Multisource Synchronization Register (`MFC_MSSync`) allows processor elements or other devices to control ordering from the main-storage domain using the MMIO register at problem-state offset x'00000'. Writing any value to this register causes the MFC to track all pending transfers targeted at the associated SPE received prior to the `MFC_MSSync` write. A read of the `MFC_MSSync` register will return a value of '1' if any transfers being tracked are still outstanding. When all the transfers are being tracked are complete, a read of this register will return a value of '0'.

To use the MMIO facility, a program must perform the following steps:

1. Write any value to the `MFC_MSSync` register.
2. Poll the `MFC_MSSync` register until a value of '0' is read.

Figure 20-2 shows the sequence of events for PPE control of multisource synchronization.

Figure 20-2. PPE Multisource Synchronization Flowchart



Use of this register is required for swapping context on an SPE. After stopping the SPU, privileged software on the PPE must prevent any new transfers from being initiated to the SPE by unmapping the associated resources. Next, privileged software must use the `MFC_MSSync` register

to ensure the completion of all outstanding transfers. This has the side-effect of ensuring that the count of the MFC Write Multisource Synchronization Request Channel, described immediately below, is ‘1’ and that the SPU Event Status (*Section 18.6.2 Procedure for Handling the Multi-source Synchronization Event* on page 474) is updated.

Note: *Frequent use of a single MFC Multisource Synchronization Facility by two or more processor elements or devices can result in a livelock condition. The livelock occurs when a read from a processor element or from a device never returns ‘0’ due to a synchronization request from other processor elements or devices.*

20.1.5.2 *MFC Write Multisource Synchronization Request Channel*

MFC Write Multisource Synchronization Request Channel (`MFC_WrMSSyncReq`) allows an SPE to control ordering from its LS-address and channel domain using its channel 9. Writing any value (‘0’ is recommended) to this channel causes the MFC to track all pending transfers targeted at itself that are received prior to the channel write. A second write to this channel will result in the SPE stalling until the outstanding transfers being tracked by the first write are complete; then, the second synchronization request will receive its response. A read of the channel count will return ‘0’ if any transfers being tracked are still outstanding. When all the transfers are being tracked are complete, a read of this channel will return a value of ‘1’.

To use the channel facility, a program must perform the following steps:

1. Write any value (‘0’ is recommended) to the `MFC_WrMSSyncReq` channel.
2. Wait for the `MFC_WrMSSyncReq` channel to become available. This can be accomplished by one of the following means:
   - Non-Event-Based:
     - Initiate a second multisource synchronization request by writing to the `MFC_WrMSSyncReq` channel. This method will block, waiting for a previous request to complete, without having to poll.
     - Poll the `MFC_WrMSSyncReq` channel count for the count to be set back to ‘1’. The `MFC_WrMSSyncReq` channel has a maximum count of ‘1’.
   - Event-Based:
     - Read the `SPU_RdEventStat` channel. This method will block, waiting for a Multisource Synchronization Event, indicating that all outstanding writes have completed.
     - Poll the `SPU_RdEventStat` channel count for the count to be set back to ‘1’.
     - Wait for an Event Interrupt, then determine the cause.
     - Loop periodically on execution of a **bisled** instruction. This instruction uses the channel count of the `SPU_RdEventStat` channel for the condition in the branch, as described in *Section 18.2.3.2 Branch Condition* on page 466.

Figure 20-4 shows the sequence of events for SPE control of multisource synchronization using monitoring methods unrelated to events. *Figure 20-4* shows the sequence of events for SPE control of multisource synchronization using event-based monitoring methods.

Figure 20-3. SPE Multisource Synchronization Flowchart (Non-Event-Based)

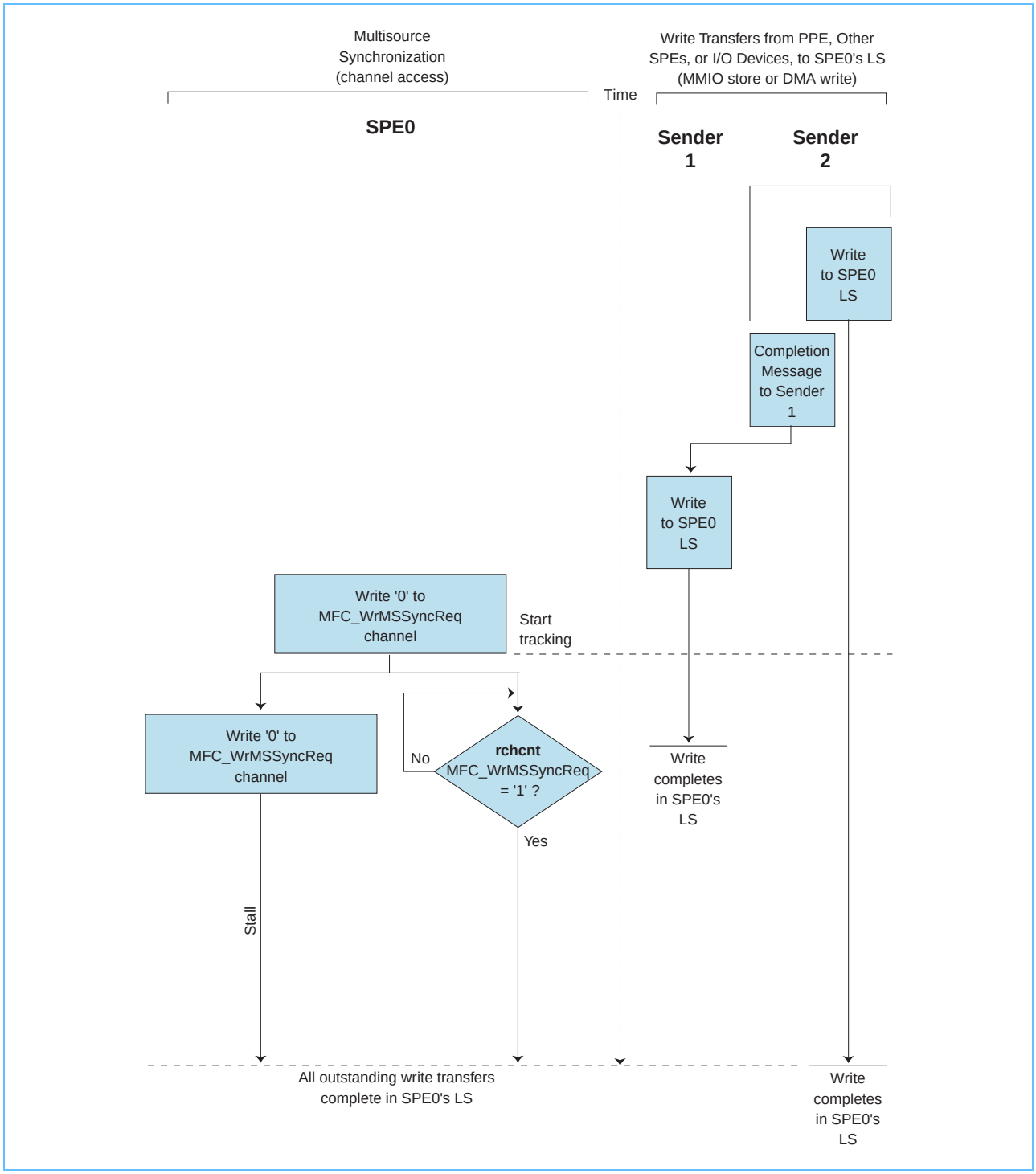
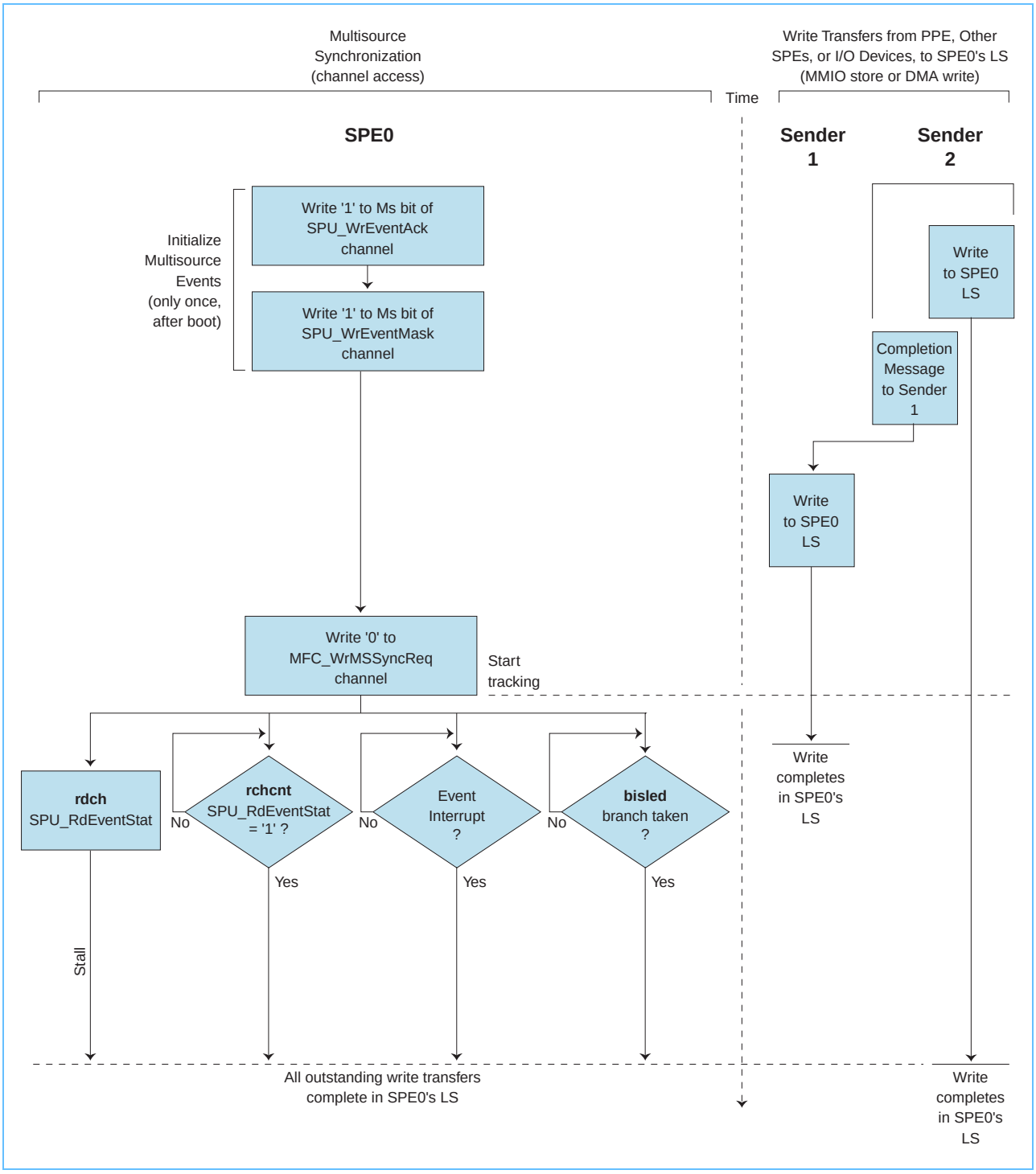


Figure 20-4. SPE Multisource Synchronization Flowchart (Event-Based)



For details about the procedures for handling a Multisource Synchronization Event, see *Section 18.6.2* on page 474.

20.1.5.3 *Multisource Synchronization Examples*

To best understand when ordering across multiple domains is required and how to use each of these facilities, consider the following examples of an I/O device storing a value of ‘1’ to an LS area, X, that is aliased in main storage, followed by a store of the value ‘2’ to a main-storage area, Y. The PPE reads main-storage area Y and stores a value of ‘3’ to another location, Z, in the LS area aliased in main storage. Multisource synchronization is required to guarantee that the value of ‘1’ that is stored in location X is ordered, with respect to the SPE, before the value of ‘3’ that is stored to location Z.

The following three examples illustrate the use of MMIO and channel multisource synchronization. The third example illustrates the use of the MMIO facility when starting an SPE. Without these facilities, the SPE is not guaranteed to receive a value of ‘1’ when reading its LS location, X.

Throughout these examples, recall that I/O transfers are always in-order.

Example 1: MFC Multisource Synchronization Register

This example demonstrates MMIO-initiated SPE multisource synchronization:

- *I/O Device:*
  a. Stores a value of ‘1’ to LS-aliased X.
  b. Stores a value of ‘2’ to main-storage location Y.
- *PPE:*
  a. Loops loading from main storage location Y until a value of ‘2’ is obtained.
  b. Stores any value to `MFC_MSSync` register.
  c. Polls `MFC_MSSync` register until a ‘0’ is obtained.
  d. Stores a value of ‘3’ to LS-aliased Z.
- *SPE:*
  a. Loops loading from LS location Z until a value of ‘3’ is obtained.
  b. Loads from LS location X, guaranteed to read a value of ‘1’

This example is provided for explanation purposes only. It is not intended as a recommendation for the use of polling loops.

Example 2: MFC Write Multisource Synchronization Request Channel

This example demonstrates channel-initiated SPE multisource synchronization:

- *I/O Device:*
  a. Stores a value of ‘1’ to LS-aliased X.
  b. Stores a value of ‘2’ to main-storage location Y.
- *PPE:*
  a. Loops loading from main storage location Y until a value of ‘2’ is obtained.
  b. Stores a value of ‘3’ to LS-aliased Z.

- *SPE:*
  a. Writes a ‘0’ to the MFC Write Multisource Synchronization Request (`MFC_WrMSSyncReq`) channel.
  b. Loads from LS location Z and obtains a 3.
  c. Wait on SPE Multisource Synchronization Event to occur by one of the following means:
     — Waiting for the channel count of the `MFC_WrMSSyncReq` channel to become ‘1’.
     — Waiting for a Multisource Sync request event.
  d. Loads from LS location X, guaranteed to read a value of ‘1’.

Example 3: MFC Multisource Synchronization Register

This example demonstrates MMIO-initiated SPE multisource synchronization when starting an SPE:

- *I/O Device:*
  a. Stores a value of ‘1’ to LS-aliased X.
  b. Interrupts PPE.
- *PPE:*
  a. Receives interrupt from I/O Device.
  b. Stores any value to `MFC_MSSync` register.
  c. Polls `MFC_MSSync` register until a ‘0’ is obtained.
  d. Starts an SPE program by writing a value of ‘1’ to the SPE Run Control register.
- *SPE:*
  a. Begins execution as a result of Step d.
  b. Loads from LS location X, guaranteed to read a value of ‘1’.

20.1.6 Scenarios for Using Ordering Mechanisms

This section includes some examples showing how the storage-ordering facilities of the CBE processor can be used.

20.1.6.1 *PPE-to-SPE Communications*

When the PPE is used as an application controller, managing and distributing work to the SPEs, the PPE typically loads main storage with the data to be processed, and then the PPE notifies the SPE by writing to either the PPE-to-SPE mailbox or one of the SPE’s signal-notification registers.

To make this feasible, it is important that the data storage be visible to the SPE prior to receiving the work-request notification. To ensure guaranteed ordering, a **lwsync** storage barrier instruction must be issued by the PPE between the final data store in memory and the PPE write to the SPE mailbox or signal-notification register.

20.1.6.2 *SPE-to-PPE Communications*

Consider a case in which the SPE places computational results into main storage, waits for the DMA transfers to complete, and then either writes to an SPE-to-PPE mailbox or performs a *stop and signal* instruction (**stop** or **stopd**) to notify the PPE that its computation is complete. Waiting for the DMA transfers to complete only ensures that the LS buffers are available for SPE reuse; it does not guarantee that data has been placed into main storage. In this case, the SPE could issue an **mfcsync** command prior to notifying the PPE. However, doing so would be inefficient. Instead, the preferred method is to have the PPE receive the notification and then issue an **lwsync** instruction before accessing any of the resulting data.

Alternatively, the SPE can perform a writeback to main storage in order to notify the PPE that the SPE's computation is complete. A writeback is a flag written to main storage that notifies the PPE that a specific event (e.g., data computation complete) has occurred. In this case, the data and the writeback must be ordered. To ensure ordering, an **mfceieio** command must be issued between the data DMA commands and the writeback DMA command.

20.1.6.3 *Implementation-Specific Ordering*

The first implementation of the *CBEA* (the CBE processor) has a fully ordered and coherent memory interface. This means that the SPE need not issue **mfcsync** or **mfceieio** commands to ensure correct ordering. Instead, ordering can be guaranteed by either waiting for command completion or using fence or barrier forms of DMA commands (see *Section 20.1.4.4* on page 564).

However, depending on these mechanisms (and omitting the use of the **mfcsync** or **mfceieio** commands) to ensure ordering may not work correctly on future CBEA-compliant implementations.

20.2 PPE Atomic Synchronization

This section describes the use of PPE atomic operations to create semaphores and mutex locks for synchronization of storage or other functions among applications. *Section 20.3* on page 587 describes the comparable subject for SPE atomic operations.

20.2.1 Atomic Synchronization Instructions

The PPE atomic synchronization instructions include the **lwarx**, **ldarx**, **stwcx.**, and **stdcx.** instructions.

The load and reserve instructions (**lwarx**, **ldarx**) load the addressed value from memory and then set a reservation on an aligned unit of real storage (called a *reservation granule*) containing the address. The CBE processor's reservation granule is 128 bytes, corresponding to the size of a PPE cache line. These 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 processor elements or memory access mechanisms.

A subsequent store conditional instruction (**stwcx**, **stdcx**) to this address verifies that the reservation is still set on the granule before carrying out the store. If the reservation does not exist, the instruction completes without altering storage. If the store is performed, bit 2 of CR0 is set to '1';

otherwise, it is cleared to ‘0’. The processor element clears the reservation by setting another reservation or by executing a conditional store to any address. Another processor element may clear the reservation by accessing the same reservation granule. A pair of load-and-reserve and store-conditional instructions permit the atomic update of a single aligned word or doubleword (only in 64-bit implementations) in memory.

A compiler that directly manages threads may use these instructions for in-line locks and to implement wait-free updates using primitives similar to compare and swap. Because locked or synchronized operations in multi-processor systems are complex, these operations are usually exposed only through calls to appropriate run-time library functions.

The **lwarx**, **ldarx**, **stwcx.**, and **stdcx.** instructions, when the load and store versions of the instructions are used together, permit atomic update of a storage location by implementing atomic primitives such as fetch-and-increment, fetch-and-decrement, and test-and-set. A programmer can choose to implement a lock, a condition variable, or a semaphore with these instructions. Examples of semaphore operations can be found in *Section 20.2.2* on page 577.

The **lwarx** instruction must be paired with an **stwcx.** instruction, and the **ldarx** instruction must be paired with an **stdcx.** instruction, with the same effective address specified by both instructions of the pair. The only exception is that an unpaired **stwcx.** or **stdcx.** instruction to any effective address can be used to clear any reservation held by the processor element. The conditional store is performed based upon the existence of a reservation established by the preceding **lwarx** or **ldarx** instruction. If the reservation exists when the store is executed, the store is performed and a bit is set in the condition register (CR). If the reservation does not exist when the store is executed, the target memory location is not modified and a bit is cleared in the CR.

The **lwarx**, **ldarx**, **stwcx.**, and **stdcx.** instructions allow software to read a semaphore, compute a result based on the value of the semaphore, store the new value back into the semaphore location only if that location has not been modified since it was first read, and determine if the store was successful. If the store was successful, the sequence of instructions from the read of the semaphore to the store that updated the semaphore appear to have been executed atomically (that is, no other processor element or mechanism modified the semaphore location between the read and the update), thus providing the equivalent of a real atomic operation. However, in reality, other processor elements may have read from the location during this operation.

At most one reservation exists simultaneously on any processor element. The address associated with the reservation can be changed by a subsequent **lwarx** or **ldarx** instruction. The conditional store is performed based upon the existence of a reservation established by the preceding **lwarx** or **ldarx** instruction.

A reservation held by the processor element is cleared (or may be cleared, in the case of the fourth and fifth items, below) by one of the following:

- The processor element holding the reservation executes another **lwarx** or **ldarx** instruction. This clears the first reservation and establishes a new one.
- The processor element holding the reservation executes any **stwcx.** or **stdcx.** instruction regardless of whether its address matches that of the **lwarx** or **ldarx**.
- Some other processor element executes a store or **dcbz** (data cache block clear to zero) to the same reservation granule (128 bytes), or modifies a referenced or changed bit in the same reservation granule.

- Some other processor element executes a **dcbtst** (data cache block touch for store), **dcbst** (data cache block store), or **dcbf** (data cache block flush) to the same reservation granule. Whether the reservation is cleared is undefined.
- Some other mechanism modifies a memory location in the same reservation granule (128 bytes). If the processor element holding the reservation modifies a Reference (R) or Change (C) bit in the same reservation granule, the lose of reservation is undefined.

Exceptions do not clear reservations. However, when a exception occurs, a **stwcx.** or **stdcx.** instruction may need to be issued to ensure that a **lwarx** or **ldarx** in the interrupted program is not paired with a **stwcx.** or **stdcx** in the new program.

See the *PowerPC Architecture* for descriptions of the atomic synchronization instructions. See *Section 20.2* on page 575 for additional memory synchronization instructions.

Note: *The PowerPC Architecture is likely to be changed in the future to permit the reservation to be lost if a* ***dcbf*** *(data cache block flush) instruction is executed on the processor element holding the reservation. Therefore* ***dcbf*** *instructions should not be placed between a load and reserve instruction and the subsequent store conditional instruction.*

The following points provide general information about the **lwarx** and **stwcx.** instructions:

- In general, the **lwarx** and **stwcx.** instructions should be paired, and the same effective address (EA) should be used for both instructions. The only exception is that an unpaired **stwcx.** instruction to any (scratch) effective address can be used to clear any reservation held by the processor element.
- It is acceptable to execute an **lwarx** instruction for which no **stwcx.** instruction is executed.
- To increase the likelihood that forward progress is made, it is important that looping on **lwarx** or **stwcx.** pairs be minimized. For example, this can be achieved by testing the old value before attempting the store; were the order reversed, more **stwcx.** instructions might be executed, and reservations might more often be lost between the **lwarx** and the **stwcx.** instructions.
- The manner in which **lwarx** and **stwcx.** are communicated to other processor elements and mechanisms, and between levels of the memory subsystem within a given processor element, is chip-implementation-dependent.
- In a multiprocessor, livelock (a state in which processor elements interact in a way such that no processor element makes progress) is possible if a loop containing an **lwarx** and **stwcx.** pair also contains an ordinary store instruction for which any byte of the affected memory area is in the reservation granule of the reservation.
- The reservation granularity is 128 bytes. The **lwarx**, **ldarx**,**stwcx.**, and **stdcx.** instructions require the effective address to be aligned.

20.2.2 PPE Synchronization Primitives

The following examples demonstrate how the PPE's atomic memory synchronization instructions, **lwarx** and **stwcx**, can be used to achieve commonly used synchronization primitives. These include atomic operations such as addition, mutexes, and condition variables. Equivalent synchronization primitives for the SPE are provided in *Section 20.2.3 SPE Synchronization Primitives* on page 580.

Additional sample PPE synchronization primitives can be found in the *PowerPC Virtual Environment Architecture, Book II*. All of the synchronization primitive examples require that all synchronization variables reside in memory marked with the memory coherency required storage control attribute (see *Section 4.2.6.7 WIMG-Bit Storage Control Attributes* on page 91 for details).

20.2.2.1 *Atomic Addition*

The atomic addition primitive atomically adds a value to a word in memory. GPR3 contains the pointer to the word in memory, GPR4 contains the value to be added to the memory word. The previous value is returned in GPR5.

```
atomic_add:
    lwarx    r5, 0, r3# load memory word with reservation
    add      r0, r5, r4# add value
    stwcx.   r0, 0, r3# store result if still reserved
    bne-     atomic_add# retry if reservation was lost
```

20.2.2.2 *Mutex Lock and Unlock*

A mutex can be used ensure mutual exclusive access to a resource or object. It is used to cause other threads to wait will while the thread holding the mutex executes code in a critical section.

The atomic synchronization instructions can be used to construct such a mutex. The mutex consists of a memory variable, pre-initialized to 0, and at least two methods, lock and unlock, that operate on the mutex variable.

The following code sample demonstrates a spin lock. The spin lock waits indefinitely until the lock can be acquired. Ideally, if the lock can not be acquired, control should be yielded to other tasks and retried at a later time.

In this example, GPR3 contains the pointer to the lock variable.

```
spin_lock:
     lwarx    r4, 0, r3# load lock variable with reservation
     cmpwi    r4, 0# is the mutex already locked
     bne-     spin_lock# keep waiting if it is
     li       r4, 1#
     stwcx.   r4, 0, r3# set the lock variable
     bne-     spin_lock# retry if store conditional failed
     isync    # delay subsequent instruction til lock acquired
```

To unlock the mutex, a simple store of zero to the mutex variable need only be done.

20.2.2.3 *Condition Variables*

A condition variable is used by a thread to make itself wait until an expression involving shared data attains a particular state. Condition variable can be used to implement other synchronization constructs such as counting semaphores.

The three primary operations on condition variables are condition wait, condition signal, and condition broadcast. This sample condition variable implementation uses a 32-bit word, pointed to by GPR3, that is atomically accessed. The word is partitioned into two half words in which the least significant half word is incremented for each conditional wait request. While the most significant halfword contains a count of the number of threads signaled. The initial value of the condition variable word is zero.

Condition wait, atomically increments the “waiting halfword”, then waits until it is signaled, that is when the “signaled” (high) halfword equals or is larger then the incremented waiting halfword. A POSIX standard condition wait function also will include a mutex unlock at the start of the condition wait and a mutex lock at the end of the condition wait. These have been omitted from the sample for brevity sake.

```
cond_wait:                      # r3 = pointer to condition variable
  _retry:
    lwarx      r5,0,r3          # atomically fetch the condition variable
    addi       r0,r5,1          # increment the waiting count
    rlwinm     r6,r5,16,16,31   # extract signaling count for cond var
    clrlwi     r7,r0,16         # extract incremented waiting count
    rlwimi     r0,r5,0,0,15
    stwcx.     r0,0,r3          # store the incremented condition variable
    bne-       _retry           # retry increment if reservation was lost
    subf       r5,r6,r7         # delta = abs(waitint_cnt-signaling_cnt)
    subf       r7,r7,r6
    cmpwi      r5,0
    bge        _wait_for_signal
    mr         r5,r7
    UNLOCK MUTEX
                                # r5 = delta, r6 = signaling count
  _wait_for_signal:             # atomically wait until signaled
    lwarx      r0,0,r3          # re-fetch the condition variable
    rlwinm     r0,r0,16,16,31   # extract current signaling count
    subf       r7,r6,r0         # compute num signaled since wait start
    subf       r0,r0,r6
    cmpwi      r7,0
    bge        _skip_next
    mr         r7,r0
  _skip_next:
    cmpw       r5,r7            # if num signaled is less than delta
    bgt-       _wait_for_signal # keep waiting
    LOCK MUTEX
```

Condition signal notifies one thread waiting on the condition variable. This is accomplished by atomically incrementing the “signaled” halfword if the waiting count is not equal to the signaled count. This will signal the oldest waiting thread to discontinue waiting.

```
cond_signal:
    lwarx      r4,0,r3          # load cond variable with reservation
    rlwinm     r5,r4,16,16,31   # compare signaled and waiting counts
    clrlwi     r6,r4,16
```

```
    cmplw      r5,r6
    beq        done              # do nothing if waiting == signaled
    addis      r4,r4,1           # increment signaled (high) halfword
    stwcx.     r4, 0, r3         # store result if still reserved
    bne-       cond_signal       # retry if reservation was lost
  _done:
```

All currently waiting threads can be signaled by making a broadcast request. A broadcast, is implemented by atomically copying the “waiting count” (low halfword) of the condition variable to the “signaled count” (high halfword) of the condition variable.

```
cond_broadcast:
    lwarx      r4, 0, r3         # load conditon variable with reservation
    rlwimi     r4, r4,16,0,15    # copy low halfword to high halfword
    stwcx.     r4, 0, r3         # store result if still reserved
    bne-       cond_broadcast    # retry if reservation was lost
```

20.2.3 SPE Synchronization Primitives

The following examples demonstrate how the MFC synchronization commands, **getllar** and **putllc**, can be used to perform synchronization primitives that are compatible with equivalent PPE primitives demonstrated in *Section 20.2.2 PPE Synchronization Primitives* on page 577. The example primitives include atomic addition, mutexes, and condition variables.

20.2.3.1 *Atomic Addition*

The atomic addition primitive atomically adds a value to a word in memory and returns the previous value. Since the granularity of the MFC synchronization commands is a cache line, this function can be easily extended to atomically alter an entire 128 byte block.

“`ea_ptr`” is a 64 bit effective address pointer to a naturally aligned 32-bit word in which “`addend`” is it to be added.

```
#include <spu_intrinsics.h>

int atomic_add_return(unsigned long long ea_ptr, int addend)
{
    int old_word;
    unsigned int offset, status;
    unsigned int ea_lo, ea_hi;
    volatile char buf[256], *buf_ptr;
    volatile int *word_ptr;

    // Determine the offset to the word within its cache line. Align
    // the effective address to a cache line bounary.
    ea_hi = (unsigned int)ea_ptr >> 32;
    ea_lo = (unsigned int)ea_ptr;
    offset = ea_lo & 0x7F;
```

```
    ea_lo &= ~0x7F;

    // Cache line align the local stack buffer.
    buf_ptr = (char *)(((unsigned int)(buf) + 127) & ~127);
    word_ptr = (volatile int *)(buf_ptr + offset);

    do {
        // Get, with reservation, the cache line containing the word to be
    // atomically added.
        spu_mfcdma64(buf_ptr, ea_hi, ea_lo, 128, 0, MFC_GETLLAR_CMD);
    (void)spu_readch(MFC_RdAtomicStat);

    // Fetch original value and add with addend.
    old_word = *word_ptr;
    *word_ptr = old_word + addend;

    // Put, conditionally the cache line containing the modified word.
        spu_mfcdma64(buf_ptr, ea_hi, ea_lo, 128, 0, MFC_PUTLLC_CMD);
    status = spu_readch(MFC_RdAtomicStat) & MFC_PUTLLC_STATUS;

    // Retry if the reservation was lost.
    } while (status);

    return (old_word);
}
```

20.2.3.2 *Mutex Lock and Unlock*

A mutex can be used ensure mutual exclusive access to a resource or object. It is used to cause other threads to wait will while the thread holding the mutex executes code in a critical section.

The MFC synchronization commands can be used to construct such a mutex. The mutex consists of a memory variable, pre-initialized to 0, and at least two methods, lock and unlock, that operate on the mutex variable.

The following code sample demonstrates a spin lock. The spin lock waits indefinitely until the lock can be acquired. The lock-line reservation lost event is used to initiate a low power stall if the mutex is currently held by another party. The input parameter, `mutex_ptr`, is a 64-bit effective address of the naturally aligned lock word.

```
#include <spu_intrinsics.h>

void lock(unsigned long long mutex_ptr)
{
    unsigned int offset, status, events, mask;
    unsigned int mutex_lo, mutex_hi;
    volatile char buf[256], *buf_ptr;
    volatile int *lock_ptr;

    // Determine the offset to the mutex word within its cache line. Align
```

```
    // the effective address to a cache line bounary.
    mutex_hi = (unsigned int)mutex_ptr >> 32;
    mutex_lo = (unsigned int)mutex_ptr;
    offset = mutex_lo & 0x7F;
    mutex_lo &= ~0x7F;

    // Cache line align the local stack buffer.
    buf_ptr = (char *)(((unsigned int)(buf) + 127) & ~127);
    lock_ptr = (volatile int *)(buf_ptr + offset);

    // Setup for use possible use of lock line reservation lost events.
    // Detect and discard phantom events.
    mask = spu_readch(SPU_RdEventStatMask);
    spu_writech(SPU_WrEventMask, 0);
    if (spu_readchcnt(SPU_RdEventStat)) {
        spu_writech(SPU_WrEventAck, spu_readch(SPU_RdEventStat));
    }
    spu_writech(SPU_WrEventMask, MFC_LLR_LOST_EVENT);

    do {
        // Get, with reservation, the cache line containing the mutex lock
   // word.
        spu_mfcdma64(buf_ptr, mutex_hi, mutex_lo, 128, 0, MFC_GETLLAR_CMD);
   (void)spu_readch(MFC_RdAtomicStat);

   if (*lock_ptr) {
       // The mutex is currently locked. Wait for for the lock line
       // reservation lost event before checking again.
       events = spu_readch(SPU_RdEventStat);
       spu_writech(SPU_WrEventAck, events);

       status = MFC_PUTLLC_STATUS;
   } else {
       // The mutex is not currently locked. Attempt to lock.
       *lock_ptr = 1;

       // Put, conditionally, the cache line containing the lock word.
       spu_mfcdma64(buf_ptr, mutex_hi, mutex_lo, 128, 0, MFC_PUTLLC_CMD);
       status = spu_readch(MFC_RdAtomicStat) & MFC_PUTLLC_STATUS;
   }

   // Retry if the reservation was lost.
    } while (status);

    // Restore the event mask
    spu_writech(SPU_WrEventMask, mask);
}
```

To unlock the mutex, a simple word store of zero to the mutex variable need only be done. This can be accomplished by either atomically writing zero to the mutex word, or alternatively, performing a DMA put to the mutex variable.

```
#include <spu_intrinsics.h>

void unlock(unsigned long long mutex_ptr)
{
    unsigned int mask;
    volatile vector unsigned int zero = (vector unsigned int){0};
    volatile void * word_ptr;
    unsigned int mutex_lo, mutex_hi;

    // Determine the offset to the word within its quadword so that our
    // source and destination addresses have the same sub-quadword alignment.
    mutex_hi = (unsigned int)mutex_ptr >> 32;
    mutex_lo = (unsigned int)mutex_ptr;

    word_ptr = (volatile void *)(&zero) + (mutex_lo & 0xF);

    spu_mfcdma64(word_ptr, mutex_hi, mutex_lo, 4, 0, MFC_PUT_CMD);

    // Wait until the DMA put completes before returning.
    mask = spu_readch(MFC_RdTagMask);
    spu_writech(MFC_WrTagMask, 1 << 0);
    (void)spu_mfcstat(MFC_TAG_UPDATE_ALL);
    spu_writech(MFC_WrTagMask, mask);
}
```

20.2.3.3 *Condition Variables*

A condition variable is used by a thread to make itself wait until an expression involving shared data attains a particular state. Condition variable can be used to implement other synchronization constructs such as counting semaphores.

The three primary operations on condition variables are condition wait, condition signal, and condition broadcast. This sample condition variable implementation uses a 32-bit word that is atomically accessed. The word is partitioned into two half words counters in which the least significant half word is incremented for each conditional wait request. While the most significant half word contains a count of the number of threads signaled. The initial value of the condition variable word is zero.

Condition wait atomically increments the “waiting halfword”, then waits until it is signaled, that is when the “signaled” (high) halfword equals or is larger then the incremented waiting halfword. A POSIX standard condition wait function also will include a mutex unlock at the start of the condition wait and a mutex lock at the end of the condition wait.

The inputs `cond_ptr` and `mutex_ptr` are 64-bit effective address to the 32-bit condition variable and mutex lock word, respectively. These are assumed to be naturally (4 byte) aligned.

```
#include <spu_intrinsics.h>

void cond_wait(unsigned long long cond_ptr, unsigned long long mutex_ptr)
{
    signed short delta, cur_delta;
    unsigned short signaled_cnt, waiting_cnt;
    unsigned int offset, events, mask, status;
    unsigned int cond_lo, cond_hi;
    volatile char buf[256], *buf_ptr;
    volatile unsigned short *cv_ptr;

    // Determine the offset to the condition variable word within its
    // cacheline. Align the effective address to a cacheline bounary.
    cond_hi = (unsigned int)cond_ptr >> 32;
    cond_lo = (unsigned int)cond_ptr;
    offset = cond_lo & 0x7F;
    cond_lo &= ~0x7F;

    // Cache line align the local stack buffer.
    buf_ptr = (char *)(((unsigned int)(buf) + 127) & ~127);
    cv_ptr = (volatile unsigned short *)(buf_ptr + offset);

    // Setup for use possible use of lock line reservation lost events.
    // Detect and discard phantom events.
    mask = spu_readch(SPU_RdEventStatMask);
    spu_writech(SPU_WrEventMask, 0);
    if (spu_readchcnt(SPU_RdEventStat)) {
        spu_writech(SPU_WrEventAck, spu_readch(SPU_RdEventStat));
    }
    spu_writech(SPU_WrEventMask, MFC_LLR_LOST_EVENT);
    // Increment the waiting half-word.
    do {
        // Get, with reservation, the cacheline containing the condition
        // variable word.
        spu_mfcdma64(buf_ptr, cond_hi, cond_lo, 128, 0, MFC_GETLLAR_CMD);
        (void)spu_readch(MFC_RdAtomicStat);

        // Increment the low half-word - waiting count.
        signaled_cnt = *(cv_ptr+0);
        waiting_cnt = *(cv_ptr+1) + 1;
        *(cv_ptr+1) = waiting_cnt;

        // Put, conditionally, the cacheline containing the condition variable.
        spu_mfcdma64(buf_ptr, cond_hi, cond_lo, 128, 0, MFC_PUTLLC_CMD);
        status = spu_readch(MFC_RdAtomicStat) & MFC_PUTLLC_STATUS;
    } while (status);

    // Unlock the mutex
    unlock(mutex_ptr);
```

```
    delta = waiting_cnt - signaled_cnt;
    if (delta < 0) delta = -delta;

    // Wait until a signal is received. I.E., high halfword of the condition
    // variable is greater than or equal to the waiting_cnt fetched when
    // the wait was established with the condition variable.

    while (1) {
        // Get, with reservation, the cacheline containing the condition
        // variable word.
        spu_mfcdma64(buf_ptr, cond_hi, cond_lo, 128, 0, MFC_GETLLAR_CMD);
        (void)spu_readch(MFC_RdAtomicStat);

        cur_delta = *(cv_ptr+0) - signaled_cnt;
        if (cur_delta < 0) cur_delta = -cur_delta;

        if (cur_delta < delta) {
            // No signal received yet. Wait until the lock line reservation
            // is lost before checking again.
            events = spu_readch(SPU_RdEventStat);
            spu_writech(SPU_WrEventAck, events);
        } else {
            // Signal received. Done waiting.
            break;
        }
    }

    // Restore the event mask
    spu_writech(SPU_WrEventMask, mask);

    lock(mutex_ptr);
}
```

Condition signal notifies one thread waiting on the condition variable. This is accomplished by atomically incrementing the “signaled” halfword count if the waiting count is not equal to the signaled count. This will signal the oldest waiting thread to discontinue waiting.

```
#include <spu_intrinsics.h>

void cond_signal(unsigned long long cond_ptr)
{
    unsigned int offset, status;
    unsigned int cond_lo, cond_hi;
    volatile char buf[256], *buf_ptr;
    volatile unsigned short *cv_ptr;

    // Determine the offset to the word within its cache line. Align
    // the effective address to a cache line bounary.
    cond_hi = (unsigned int)cond_ptr >> 32;
```

```
    cond_lo = (unsigned int)cond_ptr;
    offset = cond_lo & 0x7F;
    cond_lo &= ~0x7F;

    // Cache line align the local stack buffer.
    buf_ptr = (char *)(((unsigned int)(buf) + 127) & ~127);
    cv_ptr = (volatile unsigned short *)(buf_ptr + offset);

    do {
        // Get, with reservation, the cache line containing the condition
   // variable word.
        spu_mfcdma64(buf_ptr, cond_hi, cond_lo, 128, 0, MFC_GETLLAR_CMD);
   (void)spu_readch(MFC_RdAtomicStat);

   // If there is anyone waiting on the condition variable, increment
   // the waiting count. Else, do nothing.
   if (*(cv_ptr+0) != *(cv_ptr+1)) {
       *cv_ptr++;

       // Conditionally put the lock line containing the adjusted
       // signaled halfword.
       spu_mfcdma64(buf_ptr, cond_hi, cond_lo, 128, 0, MFC_PUTLLC_CMD);
       status = spu_readch(MFC_RdAtomicStat) & MFC_PUTLLC_STATUS;

   } else {
       status = 0;
   }
    } while (status);
}
```

All currently waiting threads can be signaled by making a broadcast request. A broadcast, is implemented by atomically copying the “waiting” count (low halfword) of the condition variable to the “signaled” count (high halfword) of the condition variable.

```
#include <spu_intrinsics.h>

void cond_broadcast(unsigned long long cond_ptr)
{
    unsigned int offset, status;
    unsigned int cond_lo, cond_hi;
    volatile char buf[256], *buf_ptr;
    volatile unsigned short *cv_ptr;

    // Determine the offset to the word within its cache line. Align
    // the effective address to a cache line bounary.
    cond_hi = (unsigned int)cond_ptr >> 32;
    cond_lo = (unsigned int)cond_ptr;
    offset = cond_lo & 0x7F;
    cond_lo &= ~0x7F;
```

```
    // Cache line align the local stack buffer.
    buf_ptr = (char *)(((unsigned int)(buf) + 127) & ~127);
    cv_ptr = (volatile unsigned short *)(buf_ptr + offset);

    do {
        // Get, with reservation, the cache line containing the condition
    // variable word.
        spu_mfcdma64(buf_ptr, cond_hi, cond_lo, 128, 0, MFC_GETLLAR_CMD);
    (void)spu_readch(MFC_RdAtomicStat);

    // Copy the wait count into the signled count.
    *(cv_ptr+0) = *(cv_ptr+1);

    // Conditionally put the lock line containing the adjusted
    // signaled halfword.
    spu_mfcdma64(buf_ptr, cond_hi, cond_lo, 128, 0, MFC_PUTLLC_CMD);
    status = spu_readch(MFC_RdAtomicStat) & MFC_PUTLLC_STATUS;
    } while (status);
}
```

20.3 SPE Atomic Synchronization

This section describes the use of MFC atomic commands to create locks for synchronization of storage or other functions among applications. *Section 20.2* on page 575 describes a comparable subject for PPE atomic operations.

The MFC for each SPE provides two main functions for the SPE: moving data between the SPE's LS and main storage, and providing synchronization between the SPE and other processor elements in the system. The MFC has an atomic unit that supports synchronization. In the CBE implementation of the *Cell Broadband Engine Architecture*, the atomic unit has a buffer that holds six 128-byte cache lines, and it dedicates four of its six cache lines to providing synchronization functions with other processing units in the system. The atomic unit does this by processing atomic MFC commands and by maintaining cache coherence when the MFC receives snoop requests from other processor elements in the system. Even though there are four cache lines dedicated to synchronization, only one reservation is maintained in each atomic unit. Four cache lines are implemented in a nested fashion (up to four deep) to provide high performance in situations where atomic updates are used for lock acquisition.

20.3.1 MFC Commands for Atomic Updates

Each MFC supports commands for atomic operations. The *get and reserve* (**getllar**) and *put conditional* (**putllc**) commands provide essentially the same functions as the PowerPC **lwarx** and **stwcx** instructions, described in *Section 20.2* on page 575. Together, these MFC commands permit atomic update of a main-storage location. In addition, the MFC supports the *put unconditional* **(putlluc**) and the *put unconditional in queued* (**putqlluc**) commands.

The **getllar**, **putllc**, and **putlluc** commands are not tagged, and they are executed immediately instead of being queued behind other DMA commands. Attempts to issue these three commands before a previous **getllar, putllc,** or **putlluc** command has completed result in an error. The

putqlluc command enables additional **putlluc** commands to be queued while the previous command executes. These four atomic-update commands can only be issued by the SPU to the MFC SPU Command Queue.

Table 20-9 lists the MFC commands for atomic updates.

Table 20-9. MFC Commands for Atomic Updates

Command	Opcode	Description
getllar	0xD0	Get lock-line and create a reservation (executed immediately)
putllc	0xB4	Put lock-line conditional on a reservation (executed immediately)
putlluc	0xB0	Put lock-line unconditional (executed immediately)
putqlluc	0xB8	Put lock-line unconditional (queued form)

20.3.1.1 *The Get Lock-Line and Reserve Command—getllar*

The get lock-line and reserve (**getllar**) command is similar to the PowerPC **lwarx** instruction, with the exception of the size and destination of the load. The **getllar** command loads 128 bytes (the PPE cache-line size) into the SPE's LS and sets a reservation visible in main storage. Issuing the **getllar** command requires an available slot in the MFC SPU Command Queue. This command is issued immediately, is not queued behind other commands, and has no associated tag. An attempt to issue this command before a previous **getllar**, **putllc**, or **putlluc** command has completed results in an error.

A read from the Atomic Command Status Channel (`MFC_RdAtomicStat`) must be performed after issuing the **getllar** command, prior to issuing another **getllar, putllc,** or **putlluc** command.

When repeating the **getllar** command to the same effective address (EA) without an intervening **putllc** or **putlluc** command, the hardware only transfers the 128 bytes of data (from EA to LS) on the first **getllar** if the reservation has not been lost to subsequent **getllar** commands. Software must avoid changing or corrupting the initial contents in the 128-byte LS area after the first **getllar**. Software must also avoid issuing the **getllar** command successively with the same EA but different LS addresses, because the data transfer only occurs on the first **getllar** command if the reservation is not lost.

20.3.1.2 *The Put Lock-Line Conditional Command—putllc*

The put lock-line conditional (**putllc**) command is similar to the operation of the PowerPC **stwcx** instruction, with the exception of the size and source of the store-conditional. This is a **put** command of 128 bytes that is conditioned on two things:

- The existence of the reservation created by the **getllar**.
- Whether the same storage location is specified by both commands.

The **putllc** command is a conditional-store operation. The store will not be successful if no reservation for the same address has been made, or if the reservation has been lost. A read of the MFC Read Atomic Command Status channel (`MFC_RdAtomicStat`) is required to determine the success or failure of this command.

20.3.1.3 *The Put Lock-Line Unconditional Command—putlluc*

The put lock-line unconditional (**putlluc**) command is similar to the operation of the **putllc** command, but the store for the **putlluc** is always performed. The **putlluc** command store is not dependent on the existence of a previously-made reservation. The store size of the **putlluc** command is 128 bytes.

20.3.1.4 *The Put Queued Lock-Line Unconditional Command—putqlluc*

The put queued lock-line unconditional (**putqlluc**) command is functionally equivalent to the put lock-line unconditional (**putlluc**) command. The difference between the two commands is the order in which the commands are performed and how completion is determined. The **putlluc** is performed immediately (not queued), and the **putqlluc** is placed into the MFC SPU Command Queue along with other DMA commands. Because this command is queued, it executes independently of any pending immediate **getllar**, **putllc**, or **putlluc** commands.

To determine if the **putqlluc** command is complete, software must wait for a tag-group completion. The **putqlluc** command requires a tag input and has an implied tag-specific fence. This command cannot be issued unless all previously issued commands with the same tag have completed.

Multiple **putqlluc** commands may be issued and pending in the MFC SPU Command Queue. All the command-queue ordering rules apply to the **putqlluc** command.

The **putqlluc** command allows software to queue the release of a lock behind commands accessing storage associated with the lock. For proper operation, the **putqlluc** command must either be within the same tag group as the commands accessing the storage, or the barrier command must be used to ensure ordering.

20.3.2 The MFC Read Atomic Command Status Channel

The MFC Read Atomic Command Status channel (`MFC_RdAtomicStat`), channel 27, contains the status of the last-completed immediate atomic update DMA command (**getllar**, **putllc**, or **putlluc**). A read from this channel before issuing an atomic command results in a software-induced deadlock. For details about the channel interface, see *Section 17* on page 441.

Software can read the count associated with this channel to determine if an atomic DMA command has completed. A value of ‘0’ is returned if an atomic DMA command has not completed. A value of ‘1’ is returned if an atomic DMA command has completed and the status is available by reading this channel. An SPE external event can also be used to determine if an atomic DMA command has completed (see *Section 18* on page 463). Regardless of the mechanism, a read from the `MFC_RdAtomicStat` channel should always follow an atomic DMA command. Performing multiple atomic DMA commands without an intervening read from the `MFC_RdAtomicStat` channel may result in an incorrect status.

This channel is read-blocking, with a maximum count of ‘1’. The contents of this channel are cleared when read. Completion of a subsequent atomic DMA command will overwrite the status of prior atomic DMA commands. *Table 20-10* shows the content of this channel.

Table 20-10. MFC Read Atomic Command Status Channel Contents

Bits	Field Name	Description
0:28	Reserved	Reserved.
29	G	Set if the **getllar** command completed.
30	U	Set if the **putlluc** command completed.
31	S	Put lock-line conditional command status: 1: Put conditional unsuccessful. The reservation was lost. 0: Put conditional successful.

20.3.3 Avoiding Livelocks

The MFC atomic synchronization commands have the benefit of allowing an SPE program to participate in atomic shared-memory updates with other processor elements in the system, including the PPE. However, care must be taken when applying these commands. Incorrect use of atomic synchronization commands can lead to large performance penalties or livelocks.

Serialize programs as infrequently as possible in order to maximize parallelism, per Amdahl's law[49]. Try to partition problems statically, and only use atomic operations. Livelock is a common problem if atomic commands are not used, and used carefully. The following sections illustrate how to avoid livelocks.

20.3.3.1 *Lock-Line Reservation Lost Event*

Use Lock-Line Reservation Lost Events instead of spin loops. This event is triggered when a get lock-line and reserve (**getllar**) command is issued and the reservation is reset due to modification of the data in the same lock-line by an outside entity. It will not be set due to a reservation reset by a local action.

Using the notification of this event allows the program to accomplish other tasks while waiting for an external modification to the lock-line data. If no other task is available, software can perform a read from the SPU Read Event Status (`SPU_RdEventStat`) channel to put the SPE into a low-power state until the lock-line data has been modified. A Lock-Line Reservation Lost Event could be used to implement a sleeping lock rather than spin locks.

For details about handling Lock-Line Reservation Lost Events, see *Section 18.6.4* on page 476.

Here is an example on how to use atomic commands with Lock-Line Reservation Lost Events:

```
/* in this example, ea and lsbuf are 128-byte aligned */

/* Disable all events */
spu_writech (SPU_WrEventMask, 0);

/* Clear phantom events */
```

49. If N is the number of processor elements, and s is the amount of time spent by one processor element on a serial parts of a program and p is the amount of time spent by one processor element on parts of the program that can be done in parallel, then the speedup is given by Speedup = 1/(s + p/N). So, speedup is inversely proportional to s, the amount of time spent in the serial parts of a program.

```
if (spu_readchcnt (SPU_RdEventStat))
  spu_readch (SPU_RdEventStat);
spu_writech(2, MFC_LLAR_LOST_EVENT_MASK);

/* set event mask */
spu_writech (SPU_WrEventMask, MFC_LLAR_LOST_EVENT_MASK);

do {
  status = 1;

  /* getllar */
  spu_mfcdma32(lsbuf, ea, 128, 0, MFC_GETLLAR_CMD);

  /* read the atomic status channel, CH 27 */
  spu_readch (MFC_RdAtomicStat);

  /* attempt to do a putllc */
  spu_mfcdma32(lsbuf, ea, 128, 0, MFC_PUTLLC_CMD);

  /* read the atomic status channel, CH 27 */
  status = spu_readch (MFC_RdAtomicStat);

  /* wait until reservation is lost */
  if (status != 0) {
    event = spu_readch(SPU_RdEventStat);
    spu_writech (2, MFC_LLAR_LOST_EVENT_MASK);
  }
} while (status != 0);

/* clean up any remnant events, as needed */
spu_writech (SPU_WrEventMask, 0);
if (spu_readchcnt (SPU_RdEventStat)) {
  event = spu_readch(SPU_RdEventStat);
  spu_writech (SPU_WrEventAck, MFC_LLAR_LOST_EVENT_MASK);
}
```

20.3.3.2 *Nesting Atomic Primitives*

Avoid nesting more than four atomic primitives. The MFC commands provides a 4-line cache for data involved in atomic updates used for lock acquisition. Nesting more than four atomic primitives may degrade performance.

20.3.4 Synchronization Primitives

The following examples demonstrate how the MFC's synchronization commands, **getllar** and **putllc**, can be used to perform synchronization primitives that are compatible with equivalent PPE primitives demonstrated in *Section 20.2.2* on page 577. The example primitives include atomic addition, mutexes, and condition variables.

20.3.4.1 *Atomic Addition*

The atomic addition primitive atomically adds a value to a word in memory and returns the previous value. Since the granularity of the MFC synchronization commands is a cache line, this function can be easily extended to atomically alter an entire 128-byte cache line.

`ea_ptr` is a 64-bit effective-address pointer to a naturally aligned 32-bit word, in which `addend` is it to be added.

```
#include <spu_intrinsics.h>

int atomic_add_return(unsigned long long ea_ptr, int addend)
{
    int old_word;
    unsigned int offset, status;
    unsigned int ea_lo, ea_hi;
    volatile char buf[256], *buf_ptr;
    volatile int *word_ptr;

    // Determine the offset to the word within its cacheline. Align
    // the effective address to a cacheline bounary.
    ea_hi = (unsigned int)ea_ptr >> 32;
    ea_lo = (unsigned int)ea_ptr;
    offset = ea_lo & 0x7F;
    ea_lo &= ~0x7F;

    // Cache line align the local stack buffer.
    buf_ptr = (char *)(((unsigned int)(buf) + 127) & ~127);
    word_ptr = (volatile int *)(buf_ptr + offset);

    do {
        // Get, with reservation, the cacheline containing the word to be
    // atomically added.
        spu_mfcdma64(buf_ptr, ea_hi, ea_lo, 128, 0, MFC_GETLLAR_CMD);
    (void)spu_readch(MFC_RdAtomicStat);

    // Fetch original value and add with addend.
    old_word = *word_ptr;
    *word_ptr = old_word + addend;

    // Put, conditionally the cacheline containing the modified word.
        spu_mfcdma64(buf_ptr, ea_hi, ea_lo, 128, 0, MFC_PUTLLC_CMD);
    status = spu_readch(MFC_RdAtomicStat) & MFC_PUTLLC_STATUS;

    // Retry if the reservation was lost.
    } while (status);

    return (old_word);
}
```

20.3.4.2 *Mutex Lock and Unlock*

A mutex can be used ensure mutual-exclusive access to a resource or object. It is used to cause other threads to wait will while the thread holding the mutex executes code in a critical section.

This mutex code sample consists of a memory variable, pre-initialized to ‘0’, and at least two methods, lock and unlock, that operate on the mutex variable. The sample demonstrates a spin lock. The spin lock waits indefinitely until the lock can be acquired. The lock-line reservation lost event is used to initiate a low-power stall if the mutex is currently held by another party. The input parameter, `mutex_ptr`, is a 64-bit effective address of the naturally aligned lock word.

```
#include <spu_intrinsics.h>

void lock(unsigned long long mutex_ptr)
{
    unsigned int offset, status, events, mask;
    unsigned int mutex_lo, mutex_hi;
    volatile char buf[256], *buf_ptr;
    volatile int *lock_ptr;

    // Determine the offset to the mutex word within its cacheline. Align
    // the effective address to a cacheline bounary.
    mutex_hi = (unsigned int)mutex_ptr >> 32;
    mutex_lo = (unsigned int)mutex_ptr;
    offset = mutex_lo & 0x7F;
    mutex_lo &= ~0x7F;

    // Cache line align the local stack buffer.
    buf_ptr = (char *)(((unsigned int)(buf) + 127) & ~127);
    lock_ptr = (volatile int *)(buf_ptr + offset);

    // Setup for use possible use of lock line reservation lost events.
    // Detect and discard phantom events.
    mask = spu_readch(SPU_RdEventStatMask);
    spu_writech(SPU_WrEventMask, 0);
    if (spu_readchcnt(SPU_RdEventStat)) {
        spu_writech(SPU_WrEventAck, spu_readch(SPU_RdEventStat));
    }
    spu_writech(SPU_WrEventMask, MFC_LLR_LOST_EVENT);

    do {
        // Get, with reservation, the cacheline containing the mutex lock
    // word.
        spu_mfcdma64(buf_ptr, mutex_hi, mutex_lo, 128, 0, MFC_GETLLAR_CMD);
    (void)spu_readch(MFC_RdAtomicStat);

    if (*lock_ptr) {
        // The mutex is currently locked. Wait for for the lock line
        // reservation lost event before checking again.
        events = spu_readch(SPU_RdEventStat);
        spu_writech(SPU_WrEventAck, events);
```

```
        status = MFC_PUTLLC_STATUS;
    } else {
        // The mutex is not currently locked. Attempt to lock.
        *lock_ptr = 1;

        // Put, conditionally, the cacheline containing the lock word.
        spu_mfcdma64(buf_ptr, mutex_hi, mutex_lo, 128, 0, MFC_PUTLLC_CMD);
        status = spu_readch(MFC_RdAtomicStat) & MFC_PUTLLC_STATUS;
    }

    // Retry if the reservation was lost.
    } while (status);

    // Restore the event mask
    spu_writech(SPU_WrEventMask, mask);
}
```

To unlock the mutex, a simple word store of zero to the mutex variable is done. This can be accomplished by either atomically, by writing zero to the mutex word, or by performing a DMA **put** to the mutex variable.

```
#include <spu_intrinsics.h>

void unlock(unsigned long long mutex_ptr)
{
    unsigned int mask;
    volatile vector unsigned int zero = (vector unsigned int){0};
    volatile void * word_ptr;
    unsigned int mutex_lo, mutex_hi;

    // Determine the offset to the word within its quadword so that our
    // source and destination addresses have the same sub-quadword alignment.
    mutex_hi = (unsigned int)mutex_ptr >> 32;
    mutex_lo = (unsigned int)mutex_ptr;

    word_ptr = (volatile void *)(&zero) + (mutex_lo & 0xF);

    spu_mfcdma64(word_ptr, mutex_hi, mutex_lo, 4, 0, MFC_PUT_CMD);

    // Wait until the DMA put completes before returning.
    mask = spu_readch(MFC_RdTagMask);
    spu_writech(MFC_WrTagMask, 1 << 0);
    (void)spu_mfcstat(MFC_TAG_UPDATE_ALL);
    spu_writech(MFC_WrTagMask, mask);
}
```

20.3.4.3 *Condition Variables*

A condition variable is used by a thread to make itself wait until an expression involving shared data attains a particular state. Condition variables can be used to implement other synchronization constructs, such as counting semaphores.

The three primary operations on condition variables are condition wait, condition signal, and condition broadcast. This sample condition-variable implementation uses a 32-bit word that is accessed atomically. The word is partitioned into two halfword counters, in which the least-significant halfword is incremented for each conditional-wait request while the most-significant halfword contains a count of the number of threads signaled. The initial value of the condition-variable word is zero.

Condition wait atomically increments the “waiting halfword”, then waits until it is signaled—that is, when the “signaled” (high) halfword equals or is larger then the incremented waiting halfword. A POSIX standard condition wait function also will include a mutex unlock at the start of the condition wait and a mutex lock at the end of the condition wait.

The inputs `cond_ptr` and `mutex_ptr` are 64-bit effective address to the 32-bit condition variable and mutex lock word, respectively. These are assumed to be naturally (4-byte) aligned.

```
#include <spu_intrinsics.h>

void cond_wait(unsigned long long cond_ptr, unsigned long long mutex_ptr)
{
    signed short delta, cur_delta;
    unsigned short signaled_cnt, waiting_cnt;
    unsigned int offset, events, mask, status;
    unsigned int cond_lo, cond_hi;
    volatile char buf[256], *buf_ptr;
    volatile unsigned short *cv_ptr;

    // Unlock the mutex
    unlock(mutex_ptr);

    // Determine the offset to the condition variable word within its
    // cacheline. Align the effective address to a cacheline bounary.
    cond_hi = (unsigned int)cond_ptr >> 32;
    cond_lo = (unsigned int)cond_ptr;
    offset = cond_lo & 0x7F;
    cond_lo &= ~0x7F;

    // Cache line align the local stack buffer.
    buf_ptr = (char *)(((unsigned int)(buf) + 127) & ~127);
    cv_ptr = (volatile unsigned short *)(buf_ptr + offset);

    // Setup for use possible use of lock line reservation lost events.
    // Detect and discard phantom events.
    mask = spu_readch(SPU_RdEventStatMask);
    spu_writech(SPU_WrEventMask, 0);
    if (spu_readchcnt(SPU_RdEventStat)) {
```

```
    spu_writech(SPU_WrEventAck, spu_readch(SPU_RdEventStat));
}
spu_writech(SPU_WrEventMask, MFC_LLR_LOST_EVENT);

// Increment the waiting half-word.
do {
    // Get, with reservation, the cacheline containing the condition
// variable word.
    spu_mfcdma64(buf_ptr, cond_hi, cond_lo, 128, 0, MFC_GETLLAR_CMD);
(void)spu_readch(MFC_RdAtomicStat);

// Increment the low half-word - waiting count.
signaled_cnt = *(cv_ptr+0);
waiting_cnt = *(cv_ptr+1) + 1;
*(cv_ptr+1) = waiting_cnt;

// Put, conditionally, the cacheline containing the condition variable.
spu_mfcdma64(buf_ptr, cond_hi, cond_lo, 128, 0, MFC_PUTLLC_CMD);
status = spu_readch(MFC_RdAtomicStat) & MFC_PUTLLC_STATUS;
} while (status);

delta = waiting_cnt - signaled_cnt;
if (delta < 0) delta = -delta;

// Wait until a signal is received. I.E., high halfword of the condition
// variable is greater than or equal to the waiting_cnt fetched when
// the wait was established with the condition variable.

while (1) {
    // Get, with reservation, the cacheline containing the condition
// variable word.
    spu_mfcdma64(buf_ptr, cond_hi, cond_lo, 128, 0, MFC_GETLLAR_CMD);
(void)spu_readch(MFC_RdAtomicStat);

cur_delta = *(cv_ptr+0) - signaled_cnt;
if (cur_delta < 0) cur_delta = -cur_delta;

if (cur_delta < delta) {
    // No signal received yet. Wait until the lock line reservation
    // is lost before checking again.
    events = spu_readch(SPU_RdEventStat);
    spu_writech(SPU_WrEventAck, events);
} else {
    // Signal received. Done waiting.
    break;
}
}

// Restore the event mask
spu_writech(SPU_WrEventMask, mask);
```

```
    lock(mutex_ptr);
}
```

Condition signal notifies one thread waiting on the condition variable. This is accomplished by atomically incrementing the “signaled” halfword count if the waiting count is not equal to the signaled count. This will signal the oldest waiting thread to discontinue waiting.

```
#include <spu_intrinsics.h>

void cond_signal(unsigned long long cond_ptr)
{
    unsigned int offset, status;
    unsigned int cond_lo, cond_hi;
    volatile char buf[256], *buf_ptr;
    volatile unsigned short *cv_ptr;

    // Determine the offset to the word within its cacheline. Align
    // the effective address to a cacheline bounary.
    cond_hi = (unsigned int)cond_ptr >> 32;
    cond_lo = (unsigned int)cond_ptr;
    offset = cond_lo & 0x7F;
    cond_lo &= ~0x7F;

    // Cache line align the local stack buffer.
    buf_ptr = (char *)(((unsigned int)(buf) + 127) & ~127);
    cv_ptr = (volatile unsigned short *)(buf_ptr + offset);

    do {
        // Get, with reservation, the cacheline containing the condition
    // variable word.
        spu_mfcdma64(buf_ptr, cond_hi, cond_lo, 128, 0, MFC_GETLLAR_CMD);
    (void)spu_readch(MFC_RdAtomicStat);

    // If there is anyone waiting on the condition variable, increment
    // the waiting count. Else, do nothing.
    if (*(cv_ptr+0) != *(cv_ptr+1)) {
        *cv_ptr++;

        // Conditionally put the lock line containing the adjusted
        // signaled halfword.
        spu_mfcdma64(buf_ptr, cond_hi, cond_lo, 128, 0, MFC_PUTLLC_CMD);
        status = spu_readch(MFC_RdAtomicStat) & MFC_PUTLLC_STATUS;

    } else {
        status = 0;
    }
    } while (status);
}
```

All currently waiting threads can be signaled by making a broadcast request. A broadcast is implemented by atomically copying the “waiting” count (low halfword) of the condition variable to the “signaled” count (high halfword) of the condition variable.

```
#include <spu_intrinsics.h>

void cond_broadcast(unsigned long long cond_ptr)
{
    unsigned int offset, status;
    unsigned int cond_lo, cond_hi;
    volatile char buf[256], *buf_ptr;
    volatile unsigned short *cv_ptr;

    // Determine the offset to the word within its cacheline. Align
    // the effective address to a cacheline bounary.
    cond_hi = (unsigned int)cond_ptr >> 32;
    cond_lo = (unsigned int)cond_ptr;
    offset = cond_lo & 0x7F;
    cond_lo &= ~0x7F;

    // Cache line align the local stack buffer.
    buf_ptr = (char *)(((unsigned int)(buf) + 127) & ~127);
    cv_ptr = (volatile unsigned short *)(buf_ptr + offset);

    do {
        // Get, with reservation, the cacheline containing the condition
   // variable word.
        spu_mfcdma64(buf_ptr, cond_hi, cond_lo, 128, 0, MFC_GETLLAR_CMD);
   (void)spu_readch(MFC_RdAtomicStat);

   // Copy the wait count into the signled count.
   *(cv_ptr+0) = *(cv_ptr+1);

   // Conditionally put the lock line containing the adjusted
   // signaled halfword.
   spu_mfcdma64(buf_ptr, cond_hi, cond_lo, 128, 0, MFC_PUTLLC_CMD);
   status = spu_readch(MFC_RdAtomicStat) & MFC_PUTLLC_STATUS;
    } while (status);
}
```

21. Parallel Programming

Programming for the CBE processor requires an understanding of parallel programming. Traditional computing platforms contain a single processor, which computes a single thread of control. High-performance computing platforms contain many processors, with potentially many threads of control. Parallel programming has become the default in many fields where immense amounts of data needs to be processed as quickly as possible: oil exploration, automobile manufacturing, pharmaceutical development and in animation and special effects studios.

These widely disparate tasks, and the algorithms for tackling those tasks, showcase the many different styles of parallel programming. Some tasks are data-centric and algorithms for working on them fit neatly into the SIMD model. Others are characterized by distinct chunks of distributed programming, and these algorithms rely on good communication models among subtasks. This chapter reviews different styles of parallel programming on the CBE processor and discusses ways to implement those different strategies. The chapter covers topics of interest to both application programmers and compiler writers.

21.1 Challenges

The key to parallel programming is to locate *exploitable concurrency* in a task. The basic steps for parallelizing any program are:

- Locate concurrency.
- Structure the algorithm(s) to exploit concurrency.
- Tune for performance.

The major challenges are:

- Data dependencies.
- Overhead in synchronizing concurrent memory accesses or transferring data between different processor elements and memory might exceed any performance improvement.
- Partitioning work is often not obvious and can result in unequal units of work.
- What works in one parallel environment might not work in another, due to differences in bandwidth, topology, hardware synchronization primitives, and so forth.

21.2 Patterns of Parallel Programming

There are several ways to express parallelism: programming language features (intrinsics and pragmas), languages with explicit support for parallelism (such as UPC and Java), tools, and APIs for using parallel-programming libraries. But with all these ways, important basic programming structures can be used to create parallel programs with good performance. These programming structures include types of algorithms and ways to synchronize access to data shared among concurrent tasks. This section describes these programming structures.

21.2.1 Terminology

The terms listed in *Table 21-1* commonly appear in textbooks and web sites focused on parallel programming. These terms describe concepts and properties that are important for efficient parallel programming.

Table 21-1. Parallel-Programming Terminology (Page 1 of 2)

Term	Definition
Amdahl's Law	Potential speedup = 1 / 1 - P, where P is the fraction of code that can be parallelized.
Asynchronous	In an asynchronous programming model, different processor elements execute different tasks without needing to synchronize with other processor elements or tasks.
Atomic	An atomic operation is uninterruptable. In parallel programming, this can mean an operation (or set of instructions) that has been protected by synchronization methods.
Bandwidth	The number of bytes per second that can be moved across a network. A program is bandwidth-limited when it generates more data-transfer requests than can be accommodated by the network.
Busy wait	Using a timed loop to create a delay. This is often used to wait until some condition is satisfied. Because the device is executing instructions in the loop, no other work can be done until the loop ends.
Critical section	A critical section is a piece of code that can only be executed by one task at a time. It usually terminates in fixed time, and another task only has to wait a fixed time to enter it. Some synchronization mechanism is required at the entry and exit of the critical section to ensure exclusive use. A semaphore is often used.
Deadlock	A deadlock occurs when two or more tasks or processor elements are stalled, waiting for each other to perform some action such as release a shared resource.
Latency	Latency is a time delay between the moment something is initiated, and the moment one of its effects begins. In parallel programming, this often refers to the time between the moment a message is sent and the moment it is received. Programs that generate large numbers of small messages are latency-bound.
Load balance	Distributing work among processor elements so that each processor element has roughly the same amount of work.
Monitor	A software monitor consists of: • A set of procedures that allow interaction with a shared resource. • A mutual-exclusion lock. • The variables associated with the shared resource. • A monitor invariant that defines the conditions needed to avoid race conditions. A monitor procedure takes the lock before doing anything else, and holds it until it either finishes or waits for a condition. If every procedure guarantees that the invariant is true before it releases the lock, then no task can ever find the resource in a state that might lead to a race condition.
Mutual exclusion	Mutual exclusion (often abbreviated to *mutex*) algorithms are used in parallel programming to avoid the concurrent use of non-shareable resources by pieces of computer code called critical sections. When several processor elements share memory, an indivisible test-and-set of a flag is used in a tight loop to wait until the other processor element clears the flag. This test-and-set ensures that when the code enters the critical region, the flag is set. When the code leaves the critical region, it clears the flag. In a spin lock or busy wait mutex, the wait loop terminates when the test finds that the flag is not set, and the wait continues if the flag is set.
Race condition	A race condition occurs when the output exhibits a dependence (usually unexpected) on the relative timing of events. The term originates with the idea of two signals racing each other to influence the output first. The term race condition also refers to an error condition where the output of a program changes as the scheduling of (multiple) processor elements varies.

Table 21-1. Parallel-Programming Terminology (Page 2 of 2)

Term	Definition
Semaphore	A semaphore is a protected variable (or abstract data type) and constitutes the classic method for restricting access to shared resources (for example, shared memory) in a parallel programming environment. There are two operations on a semaphore: V (sometimes called up) and P (sometimes called down). The V operation increases the value of the semaphore by one. The P operation decreases the value of the semaphore by one. The V and P operations must be atomic operations.
Starvation	Starvation occurs when a task tries to access some resource but is never granted access to that resource.
Synchronous	Coordinated in time among tasks, such as when one task notifies a second task about some state change, which the second task receives when it is polling for such notification.
Synchronization	Synchronization enforces constraints on the ordering of events occurring in different processor elements.
Task	A unit of execution.
Thread	A fundamental unit of execution in many parallel programming systems. One process may contain several (or dozens or hundreds) of threads.

21.2.2 Finding Parallelism

The first, vital step in parallelizing any program is to consider where there might be exploitable concurrency. Time spent analyzing the program and its algorithms and data structures will be repaid many-fold in the implementation and coding phase. In other words, by no means immediately start to code a program to take advantage of this or that parallel programming model. Spend time understanding the data flow, data dependencies, and functional dependencies.

The most important question is: Will the anticipated speedup from parallelizing a program be greater than the effort to parallelize a program, which includes any overhead for synchronizing different tasks or access to shared data?

The second question is: Which parts of the program are the most computationally intensive? It is worthwhile to do initial performance analysis on typical data sets, to be sure the hot spots in the program are being targeted.

When you know which parts of the program can benefit from parallelization, you can consider different patterns for breaking down the problem. Ideally, you can identify ways to parallelize the computationally-intensive parts:

- Break down the program into tasks that can execute in parallel.
- Identify data that is local to each subtask.
- Group subtasks so that they execute in the correct order.
- Analyze dependencies among tasks.

In this context, a *task* is a unit of execution that is enough to keep one processor element busy for a significant amount of time, and that performs enough work to justify any overhead for managing data dependencies. Key elements to examine are:

- Function calls.
- Loops.
- Large data structures that could be operated on in chunks.

Questions to ask about data-sharing include:

- Do all processor elements update shared data?
- Does one task need another's data?
- Are there any race conditions, in which data might be read either before or after it is written in another task?
- Is there too much synchronization overhead to justify the parallelism?
- Are interactions among tasks synchronous or asynchronous?
- Are the processing advantages worth the data-communication costs, including latency, bandwidth, and amount of data that needs to be transferred?

Main organizing principles include:

- By tasks (including control flow between tasks and the amount of work in different tasks).
- By data decomposition.
- By flow of data.

21.2.3 Strategies for Parallel Programming

21.2.3.1 *Task Parallelism*

Some programs are embarrassingly parallel. For example, ray-tracing routines fit well into task-parallel designs. Such programs have, inside them, many tasks that are completely independent. These programs can be easily mapped onto standard parallel programming models.

The two essential questions are:

- Are the tasks independent enough that managing dependencies does not consume too much time?
- Can the tasks be load-balanced among the CBE processor elements?

21.2.3.2 *Data Partitioning*

Some programs operate on data that can be broken down into chunks that can be operated on independently. This approach makes sense when the most computationally-intensive part of the program is centered on manipulating a large data structure. Use this approach when the chunks of data are operated on in the same way. Arrays of data, and recursive data structures like trees, are often good candidates for this style of data decomposition.

21.2.3.3 *Task Grouping*

Sometimes groups of tasks—several sets of operations on data—can be pulled together to form the basis of parallelism. Combining a group of tasks can create enough work to justify assigning it to a single processor element. This approach can also reduce the amount of synchronization between different groups, with each group being scheduled to execute as a single task. This approach also makes sense when one task shares data dependencies with another task. Those tasks may be candidates for grouping.

21.2.3.4 *Divide and Conquer*

A divide and conquer algorithm works by recursively breaking down a problem into two or more sub-problems of the same (or related) type, until these become simple enough to be solved directly. The solutions to the sub-problems are then combined to give a solution to the original problem. This strategy is often used in mathematical problems, such as fast Fourier transform (FFT) or Cholesky matrix decomposition. The parallelism is easy to use: since the sub-problems are solved independently, they can be computed in parallel.

21.2.3.5 *Pipelining*

Sometimes the overall computation involves feeding data through a series of operations. Graphics applications often fall into this category. The key here is to express clearly any ordering constraints—which operations must occur before any other. Pipelining can be expressed at different levels: at the SIMD or vector-processing level, as well as at a higher, algorithmic level. The key to parallelism is assigning each stage of the pipeline to a different processor element, and managing data flow among the processor elements.

21.2.3.6 *Event Parallelism*

Event parallelism concerns a group of independent tasks that are interacting, but in a somewhat irregular fashion. This style of programming often relies on asynchronous communication, in which a task sends an event but does not wait for a response. In a shared-memory system, the program might use a queue to represent message-passing among the tasks. This requires safe, concurrent access to mutex variables. One challenge is avoiding deadlock, in which one task waits for an event that will never arrive. Another challenge is load-balancing the tasks across processor elements.

21.2.3.7 *Master and Slave*

The master and slave approach is similar to task parallelism. One master processor element might farm out chunks of works to slave processor elements and wait for these slave tasks to return. This approach works well when you have variable and unpredictable workloads, and simple parallel loops are not enough. The master task is in charge of load-balancing among the several slave tasks. A shared queue is a good way to manage the workload. This approach has good scalability if the number of tasks greatly exceeds the number of slave processor elements.

21.2.3.8 *Fork and Join*

The fork and join approach works well when the number of parallel tasks varies as the program runs, and structures such as parallel loops are not powerful enough to contain each task's execution. Tasks are created dynamically: that is, they are *forked*. When tasks finish, they return their status to the parent: that is, they are *joined*. A one-to-one mapping of tasks onto processor elements often works well, but this is dependent on the number of tasks and the number of processor elements.

21.3 Steps for Parallelizing a Program

Now that we have introduced the common strategies for parallel programming, this section suggests ways in which you can apply these strategies to a program.

21.3.1 Step 1: Understand the Problem

- *Performance Objectives*—Is the objective to perform the computation as fast as possible, using all available resources? Or is absolute performance not critical, and is efficiency more important because the algorithm must be run simultaneously with other problems?
- *Type of Computation*—Does the problem contain small kernels of computation that can be partitioned and loaded onto separate SPEs? Also, because of the heterogeneous architecture, it is important to understand what code sections must be performed on the PPE (such as system calls) or may be faster on a PPE (such as code dominated by unpredictable branches), and what code sections are likely to be faster on an SPE.
- *Data Types*—Is the fundamental data type fixed-point or floating-point? What is the precision—byte, halfword, word, or doubleword? The answers to these questions may determine which computational resources can be used. For example, double-precision floating-point is not supported by the PPE's SIMD Vector/SIMD Multimedia Extension instructions, although it is supported by the PPE's standard *PowerPC Architecture* instructions and the SPU instructions.(The SPEs, however, are optimized for single-precision floating-point operations.)
- *Data-Access Patterns*—Is data randomly accessed or sequentially accessed? If randomly accessed, what is the size of the data accesses? Small data accesses can negatively affect the memory bandwidth so that alternative data organizations may be warranted. Understanding data-access patterns includes understanding what data is read-only, write-once, accessed by a single task in a particular parallel region, or accessed only by a specific subset of tasks.

21.3.2 Step 2: Choose Programming Tools and Technology

- *System Configuration*—Is this application going to be deployed on a networked cluster of CBE processors, on a shared-memory system of CBE processors, on a single CBE processor, or on only one (or a few) of a CBE processor's processing elements?
- *Availability*—What tools are available? Because the CBEA is a new architecture, new tools are being developed all the time.
- *Quality and Reliability*—Are the tools stable and of high-enough quality for the development needs?
- *Programmer Skills*—Are the programmers experienced in the use of tools and languages? Can the project schedule absorb the learning curve associated with using new programming tools or languages?
- *Schedule Objectives*—How much time is available to develop the software?
- *Performance Objectives*—What are the system performance objectives? Can software be quickly developed now and optimized over time?

21.3.3 Step 3: Develop High-Level Parallelization Strategy

The key to maximizing system performance on a CBE system is choosing the optimal parallelization strategy. Depending on the system configuration, multiple strategies may be required—for example, a high-level parallelization strategy and a moderate-level parallelization strategy. A high-level strategy is useful when partitioning the problem across systems without shared-memory access. A moderate-level strategy is useful when partitioning the problem across processing elements that have high-speed, shared-memory access.

The key to nearly all successful parallelization strategies is *minimizing synchronizing events* between the computational elements. See *Section 20 Shared-Storage Synchronization* on page 551. One effective method for the CBE processor is to partition the data for independent computation across parallel SPE threads of execution. Each SPE thread operates as an independent stream processor—streaming in blocks of input data, performing the necessary computation, and streaming out the results. Multibuffering techniques (see *Section 24.1.2* on page 682) can be employed to ensure simultaneous computation and data transfer.

The work (data) can be allocated among the SPEs in one of three methods:

- *Algorithmically Assigned*—For example, if *w* work is partition among *n* SPEs, then each SPE is assigned *w*/*n* amount of work. This solution works well for simple algorithms in which the computation patterns are deterministic.
- *Self-Managed*—Making SPEs autonomously arbitrate for work is effective at load-balancing parallel computation when computation is not easily predictable.
- *Mastered*—In this method, one of the processing elements (typically the PPE) serves as a master distributing work among the available SPEs. Work is communicated to the SPEs either by using a work queue in main storage or by using one of the hardware communication features—mailboxes (*Section 19.6* on page 531) or signal-notification registers (*Section 19.7* on page 542).

There is a set of problems in which an SPE's local store (LS) is insufficient to hold both the working data set and all the code simultaneously. In these situations, it may be effective to stream the code to the data by sequencing a set of code modules over a single set of LS data.

21.3.4 Step 4: Develop Low-Level Parallelization Strategy

Language extensions, like vector intrinsics, and pragmas, like OpenMP's `parallel do`, can be used to direct the compiler in its SIMDization phase. Identify data structures that can be operated on using vector instructions, and annotate the code accordingly. See *Section 22* on page 619 for more information on SIMD Programming, and see *Section 21.4.1* on page 608 for details on programmer-directed SIMDization.

21.3.5 Step 5: Design Data Structures for Efficient Processing

After the basic low-level parallelization strategy is identified, the data structures must be developed for efficient processing. This includes:

- *Format*—Organize structures and arrays according to the SIMD format chosen in Step 4 (*Section 21.3.4*). Consider padding SOA-formatted data (*Section 22.1.1.1* on page 620) to an even multiple of quadwords, so that partial quadword processing need not be used, thus improving performance.

- *Quadword Align*—Quadword-align arrays and structures to ensure correct alignment for transferring data from main storage to an LS, and to ensure correct alignment for LS loads and stores. Some data structures may also be aligned on a cache-line boundary to maximize bus bandwidth.
- *ABI Differences*—Pointers and long variables may be of different sizes, depending upon whether the PPE is compiled according the 32-bit ABI or 64-bit ABI. For example, the 32-bit ABI may be ILP32 (that is, 32-bit integers, longs, and pointers). The 64-bit ABI may be LP64 (that is, 64-bit longs and pointers). The SPEs are always ILP32, with DMA support for 32- and 64-bit effective-address pointers.
- *Synchronization Variables in Independent Cache Lines*—Place synchronization variables in their own, independent 128-byte cache lines to minimize atomic-reservation thrashing.
- *Optimize Layout*—Tailor layout of shared data structures for parallel computation—that is, to provide spatial locality. For example, structure data so that (as much as possible) data items used locally within a processor are not interspersed with data items shared by multiple processor elements.

When data structures are shared, one or more tasks might modify the data, and one or more tasks might access the modified data. Explicitly managing shared data structures is one of the most error-prone aspects of parallel programming. Try to reduce the size of the critical section. Use a mutual-exclusion protocol to access the shared data: mutex locks, synchronization blocks, semaphores, or nested locks. Consider partially replicating shared data, so that many tasks can update the data without need to synchronize with one another. But don't prevent concurrent operations on shared data that could safely have taken place at the same time. Also, prevent bottlenecks, in which several tasks are waiting to access the same areas of shared data at the same time.

21.3.6 Step 6: Iterate and Refine

The next step is to evaluate system performance and determine the effectiveness of the parallelization strategy. Various performance-analysis techniques can be employed to identify the performance bottlenecks. Based upon these findings, the strategies devised in Step 3 (*Section 21.3.3* on page 605) and Step 4 (*Section 21.3.4* on page 605) may need to be revisited.

21.3.7 Step 7: Fine-Tune

After the basic parallelization strategy is established in the above steps, the software should be fine-tuned. Generally, this entails fine-tuning only the critical code sections identified by either programmer knowledge or hot-spot analysis. Programmer knowledge of PPE and SPE programming, as well as previous experience with the code-development tools, will influence the degree of additional fine-tuning that is necessary.

Fine-tuning may include one or more of the following:

- *Branches*—Eliminate branches by exploiting the select-bits instruction (see *Section 24.3.2* on page 689). Reduce mispredicted branches by using feedback-directed optimization, software branch-prediction tables, or programmer-directed branch prediction.
- *Inline*—Inline frequently-called functions, to eliminate call-linkage overhead and provide additional instructions for improved scheduling and latency-hiding. See *Section 24.3.1* on page 689. Most compilers give users control over what and how much to inline.

- *Loops*—Unroll loops to reduce loop iterations and provide additional instructions for improved instruction scheduling. See *Section 24.3.1* on page 689. Loop-unrolling may not prove fruitful for algorithms in which dependencies exist between loop iterations. Be careful of the interaction of manual loop-unrolling with compiler SIMDization; check the results after compiling. Also, loop unrolling may not always be beneficial, due to constrained LS size in an SPE.
- *False Dependencies*—Restructure code to eliminate false-dependency stalls. False-dependency stalls are instruction dependencies that exist only because of the order in which the operations were scheduled.
- *Dual-Issue*—Change instruction mix to improve dual-issuing. Frequently, there are multiple ways an operation can be performed. The multiple solutions may use instructions on different execution pipelines. Choosing the solution that uses instructions executed on an under-used pipeline can result in improved performance. Explicit instruction control can be achieved when programming in intrinsics.
- *Memory-Bank Access*—Refine data organization to ensure better memory-bank access patterns. Many algorithms have regular, power-of-two, data-access patterns. Because memory banks are organized on powers of two, this can result in multiple processor elements accessing the same memory banks at the same time. This, in turn, results in less-then-theoretical memory bandwidth. Offsetting data buffers or algorithmically changing the access patterns can improve performance on algorithms that are memory bound.
- *Inefficient Operations*—Eliminate inefficient operations based upon known input constraints. For example, integer multiply can be optimized if one or more of the operands is known to be 16 bits or less. See *Section 24.7* on page 705.

21.4 Levels of Parallelism in the CBE Processor

The CBE processor provides a foundation for many levels of parallelization. Starting from the lowest, fine-grained parallelization—SIMD processing—up to the highest, course-grained parallelization—networked multiprocessing—the CBE processor provides many opportunities for concurrent computation. The levels of parallelization include:

- SIMD processing.
- Dual-issue superscalar microarchitecture.
- Multithreading.
- Multiple execution units with heterogeneous architectures and differing capabilities.
- Shared-memory multiprocessing.
- Networked distributed-memory multiprocessing.

The shared-memory model is key to data transfer and program execution across the PPE and the SPEs. Compilers and other utilities can mediate the partitioning of code and data and orchestrate any data movement implied by this partitioning. A variety of compiler techniques can be used to exploit the performance potential of the SPEs and to enable the multi-level heterogeneous parallelism supported by the CBE processor. For example, see *Section 21.5.3* on page 611 and *Section 21.5.4* on page 613.

21.4.1 SIMD Parallelization

Section 22 on page 619 describes SIMD parallelization techniques. Both the PPE and the SPEs are capable of SIMD computation. Compilers can support programmer-directed parallelism by means of programmer-inserted intrinsics and pragmas. Compilers can also provide auto-SIMDization that generates vector instructions from scalar source code for both the PPE and the SPEs. Auto-SIMDization minimizes the overhead due to misaligned data streams and can be tailored to handle many of the data structures and code structures found in multimedia applications.

21.4.2 Superscalar Parallelization

The PPE and SPEs have multiple, parallel execution units and are capable of executing two instructions per clock. The PPE execution units are described in *Section 2.1* on page 46. The SPE execution units are described in *Section 3.1* on page 63. Dual-issue success depends upon the instructions being issued, their address, and the state of the system during execution.

A dual-issue instruction pair consists of two instructions on an aligned doubleword address. *Section A.5* on page 750 describes the dual-issue rules for the PPE's Vector/SIMD Multimedia Extension instructions. *Section B.1.3* on page 768 describes the dual-issue rules for SPE instructions.

21.4.3 Hardware Multithreading

The PPE supports two simultaneous threads of execution in hardware, so the PPE can be viewed as a two-way multiprocessor with shared dataflow. This gives PPE software the effective appearance of two independent processing units. The performance of the two threads is limited, however, because they must share resources such as the L1 and L2 caches and they must reside in the same logical partition. For more details on multithreading, see *Section 10 PPE Multithreading* on page 295.

Programmers typically think of multithreading in terms of application programs and the thread libraries they may use. CBE thread libraries can enable multithreading at the application level.

21.4.4 Multiple Execution Units

Each of the CBE processor's nine processor elements provide independent computation and can be considered asymmetric threads of execution. All processor elements have access to the coherent main storage for shared-memory multiprocessing. The SPE mailboxes and signal notification registers support parallel-processing message-passing.

Compilers can parallelize and partition a single source program across the PPE and SPEs, guided by user directives. Compilers can also optimize data transfer, allowing a single SPE to process data that far exceeds the LS's capacity. The SPE may also use code that exceeds the size of LS. Compilers can schedule necessary DMA transfers so that they overlap ongoing computation.

In addition to the nine processor elements, each of the eight MFCs serve as independent DMA controllers capable of supporting up to 16 concurrent transfers for a total of 128 such concurrent transfers. The large number of concurrent transfers, in conjunction with a uniform distribution of memory transfers across both XDR memory channels and all 16 XDR memory banks, makes near-theoretical peak memory throughput possible.

21.4.5 Multiple CBE Processors

Multiple-instruction, multiple-data (MIMD) parallelization can be supported on one CBE processor or on multiple CBE processors. This section reviews two common methods of interconnecting multiple CBE processors.

21.4.5.1 *Shared-Memory Processing*

Two CBE processors can be directly connected by means of the Cell Broadband Engine Interface (BEI) Unit, which manages data transfers between the EIB and I/O devices as shown in *Figure 7-1* on page 161. The BEI supports two Rambus FlexIO I/O interfaces. One of the two interfaces supports only a non-coherent I/O Interface (IOIF) protocol, which is suitable for I/O devices. The other interface is software-selectable between the non-coherent interface and the fully coherent Cell Broadband Engine Interface (BIF) protocol—the EIB's native internal protocol—which coherently extends the EIB to another device that can be another CBE processor.

Figure 7-2 on page 163 shows a two-way shared-memory processing configuration, and *Figure 7-3* on page 164 shows a four-way configuration. Configurations with more than two CBE processors require a BIF-protocol switch, and such switches can be connected together to support very large CBE multiprocessing configurations.

The BIF protocol supports shared, addressable I/O memory. I/O addresses can be optionally translated from effective addresses into real addresses using the I/O address-translation mechanism described in *Section 7.4* on page 176.

The programming of shared-memory systems, whether on one CBE processor or multiple CBE processors, is often done using either OpenMP, Pthreads, or Unified Parallel C (UPC). See *Section 21.5.3* on page 611 for details.

21.4.5.2 *Distributed-Memory Processing*

Multiple CBE processors and/or CBE-processor complexes can be loosely clustered in a distributed-memory configuration. Such a system consists of CBE processors, memory associated with each CBE processor, and an interconnection network. Because such systems lack shared memory between CBE processors, data that is needed by more than one CBE processor must be explicitly sent from CBE processor to CBE processor through the interconnection network.

Distributed-memory processing systems are typically programmed in one of three methods:

- Using a proprietary distributed multiprocessing framework.
- Using the Message Passing Interface (MPI) standard. MPI is widely available and was designed for high-performance communication on both massively parallel architectures and clustered distributed-memory systems.

- Using grid computing services, consisting of a set of standards and protocols. For example, the Open Grid Services Architecture (OGSA) enables communication across heterogeneous, geographically dispersed environments.

21.5 Tools for Parallelization

Given an understanding of the patterns and steps of parallelization, and an understanding of what parallelism is available on the CBE processor, how can parallelism be implemented efficiently? This section describes some tools for creating parallel programs.

21.5.1 Language Extensions: Intrinsics and Directives

A broad set of intrinsic extensions for C and C++ is defined in the *SPU C/C++ Language Extensions* specification and summarized in *Appendix B.2* on page 772. Compiler intrinsics function as fast shorthand for expressing assembly-level instructions. Compilers that support the intrinsics will likely also support vector data types. Each data type in C or C++ can be prepended by the keyword, `vector`, to form a 128-bit vector of two double words, four words, eight halfwords, or 16 bytes.

Intrinsics are very useful when you are manually SIMDizing your code. If you are coding at this level, be sure to place small structures that must be accessed by mutual exclusion in their own cache line, so they can be operated on by the intrinsic equivalents of the MFC atomic commands, **getllar**, **putllc**, **putlluc** and **putqlluc** (*Section 19.2.1 DMA Commands* on page 508). However, it may not be appropriate to use intrinsics in all cases. For example, if you are relying on the compiler to perform SIMDization, it is best to let the compiler select the appropriate assembly-level instructions.

The major advantage of using intrinsics is that they provide full control over the handling of data alignment and the selection of SIMD instructions, yet still provide the benefits from the compiler's other optimizations such as loop unrolling, function inlining, scheduling, and register allocation. In addition, you can normally rely on the compiler to generate scalar code, such as address, branch, and loop code. Such code is error-prone when coded manually in assembly language. Furthermore, it is much easier to modify code that uses intrinsics than it is to maintain assembly code.

In addition to the hundreds of intrinsics at your disposal, a compiler may provide the special directives that are summarized in *Appendix B.2.5* on page 779. For example, branch prediction can be helped with feedback-directed optimization:

```
int __builtin_expect(int exp, int value)
```

This instructs the compiler that `exp` is expected to equal `value`.

The `__align_hint` directive enables compiler analysis and optimization for pointers. It provides compilers with information that is needed for auto-SIMDization:

```
__align_hint(ptr, base, offset)
```

Here, `ptr` points to data with the base alignment of `base` and an offset from `base` of `offset`.

The most important tool for efficient, highly optimized low-level assembly code is your compiler. Refer to its documentation for a description of compiler optimizations that you can enable for your program.

21.5.2 Compiler Support for Single Shared-Memory Abstraction

Manually partitioning code and data among the PPE and the SPEs, and manually managing transfers of code and data between main storage and SPEs' LSs, are great burdens on a programmer. A programmer is accustomed to viewing a computer system as possessing a single addressable memory in which all program data reside.

In the CBE processor, the LSs, which are directly addressable by their respective SPUs, are memories separate from the vastly larger main storage. Each SPE can initiate DMA transfers between its LS and main storage. The compiler, under certain conditions, can also initiate DMA transfers by explicitly inserting DMA commands. Moreover, there are many optimizations that the compiler can perform to optimize these data transfers, especially when memory references are regular. The compiler can attempt to abstract the concept of separate memories by allocating SPE program data in main storage and automatically managing the movement of this data between its home location and a temporary location in an LS. If this movement is done to satisfy the demands of an executing SPE program, and if the resulting buffers are organized in such a way as to permit reuse, this is referred to as a *software cache*. See *Section 21.5.4* on page 613 for details about software caches.

21.5.3 OpenMP Directives

Many compilers support some set of OpenMP directives. OpenMP is a parallel programming model for shared-memory systems. Pioneered by Silicon Graphics, Inc. (SGI), it is now embraced by most of the computing industry and is exhaustively documented at http://www.openmp.org. The book *Parallel Programming in OpenMP*, by Chandra et al., is another good source, as is *Parallel Programming with OpenMP* at http://www.osc.edu/hpc/training/openmp.

OpenMP directives are instructions to the compiler to perform tasks in parallel. In Fortran, they are expressed as source-code comments; in C and C++, they are expressed as a `#pragma`. All OpenMP directives in C and C++ are prefixed by, `#pragma omp`. A typical OpenMP directive looks like this one, which says that individual iterations of the loop are independent and can be executed in parallel:

```
#pragma omp parallel for
    for (int i = 0; i < 1000; i++)
        big_calc(x[i]);
```

OpenMP includes a small set of runtime library routines and environment variables. These routines are designed to be thread-safe. The environment variables can be used to control thread behavior such as spin, yielding, and thread binding. Also, each thread created by OpenMP has its own stack that is off the heap. The compiler often has flags to control the default stack size for each thread. Without this functionality, a program could easily run out of memory.

The basic execution model for OpenMP is the fork and join model. OpenMP provides two parallel control constructs. The first control construct is the `parallel` directive. It encloses a block of code and causes new threads of execution to be spun off to execute that block of code. The second control construct is a mechanism that divides work among a group of already-existing parallel threads.

Variables can be declared `shared`, which means they have a single storage location during the execution of a parallel section. In contrast, a variable can be declared to be `private`. It will have multiple copies: one within the execution context of each thread. There are different flavors of `private`, including `threadprivate`, `firstprivate`, and `lastprivate`. The latter two include an element of shared storage behavior. Finally, a variable can be declared to be a `reduction` variable. Reduction variables have both private and shared storage behavior. A common example is the final summation of temporary local variables at the end of a parallel construct.

OpenMP provides mutual exclusion and event synchronization as ways for multiple OpenMP threads to communicate with each other. The directives `critical` and `barrier` can be used.

OpenMP allows programmers to specify loops, loop nests, or code sections that may be executed in parallel. This, of course, relies heavily on the programmer doing the initial data and task analysis, to make sure that sections marked for parallelization are safe to parallelize. Although OpenMP is a shared memory model, it can also be implemented for distributed memory systems. In the case of the CBE processor, an SPE's LS is not shared memory, but a compiler can simulate shared memory for SPEs using a combination of a software-managed cache (*Section 21.5.4* on page 613) and DMA-transfer optimizations (*Section 24.1.2* on page 682).

An OpenMP implementation typically uses a runtime library. The runtime library includes functions for initialization, work distribution, and synchronization of data, as well as control flow. The compiler can insert calls to runtime library functions appropriate to the OpenMP directives contained in the source code. The OpenMP *master thread* runs on the PPE and uses the runtime library to distribute work to the SPEs. The master thread itself partakes in all work-sharing constructs. This thread also handles all OS service requests. The PPE runtime library can include the facility to spawn new SPE threads and terminate them. When a new SPE thread is spawned, it can continuously loop, waiting for the PPE to assign work items to it. A work item specifies a handle that determines the function to execute, any input parameters, and if needed, the current address of the stack in main storage.

Such a runtime library requires communication between the PPE and the SPEs to coordinate execution. SPEs can use explicit DMA commands to read work items assigned to them from a circular queue that is shared with the PPE. The *Cell Broadband Engine Architecture* also includes more efficient communication channels in the form of signal notification and mailbox facilities. The PPE can use asynchronous signals to inform an SPE that there is work available, or that it should terminate. The SPEs can use the mailbox facility to update the PPE on the status of their execution.

Several factors complicate the implementation of OpenMP on the CBE processor. There are two distinct instruction sets. Certain code sections may be executed on both the PPE and the SPEs. Since the SPEs are not designed to run an operating system, certain code sections such as system calls must execute on the PPE. The compiler must identify data in main storage that is accessed in SPE code sections, and it must insert DMA commands at appropriate points to transfer this data to and from the SPE LSs. The SPE LSs are limited to 256 KB, and this space

contains both the code and data that is executing on the SPE. The compiler may need to split SPE code into multiple partitions if it does not fit in the limited SPE LS (see *Section 14* on page 391 for some alternatives).

The challenge for the compiler is to efficiently manage all this complexity while still enabling the performance potential of the CBE processor. To do so, the compiler must:

- Minimize the amount of data that needs to be transferred using DMA commands.
- Align some data in main storage for efficient DMA transfers.
- Issue DMA commands early in the code to overlap computation and communication.
- Carefully choose the size of code partitions and individual data transfers.

For details about OpenMP in the context of SIMD programming, see *Section 22.4.1* on page 664.

Here are some things a programmer can do to get better performance when using a compiler that supports OpenMP:

- Choose intelligent values for OpenMP attributes like scheduling and number of threads.
- Minimize use of pointers in parallel code, to enable more precise compiler analysis.
- Lay out shared data structures carefully, paying attention to locality of data in the context of parallel execution.
- Use alignment and other intrinsics to guide compiler analysis.

21.5.4 Compiler-Controlled Software Cache

An SPE's local store (LS), which is filled from main storage using software-initiated DMA transfers, can be regarded as a software-managed cache (or simply *software cache)*. Although most processors reduce latency to memory by using hardware caches, an SPE uses its DMA-filled LS. The approach provides a high degree of control, but it is advantageous only if the DMA transfer-size is sufficiently large and the DMA command is issued well before the data is needed, because the latency and instruction overhead associated with DMA transfers exceeds the latency of servicing a cache miss on the PPE.

When compiling SPE code, the compiler can identify references to data in main storage that have not been optimized using explicit DMA transfers, and the compiler can insert code to invoke the software-managed cache mechanism before each such reference. The call to the software cache takes a main-storage address and returns an address in the SPE LS that contains the corresponding data. The SPE code is generated by a compiler code generator that is different from the one that generates PPE code. Symbols that refer to main storage will have space allocated to them when the PPE code is compiled and linked. The SPE code does not have access to these addresses until it is linked with the PPE code at the very end. During compilation, the compiler can create new symbols—local Table Of Contents (TOC) entries—in the SPE code as placeholders for the main storage addresses. The correct values for these symbols can be filled in when the PPE and SPE code sections are linked together.

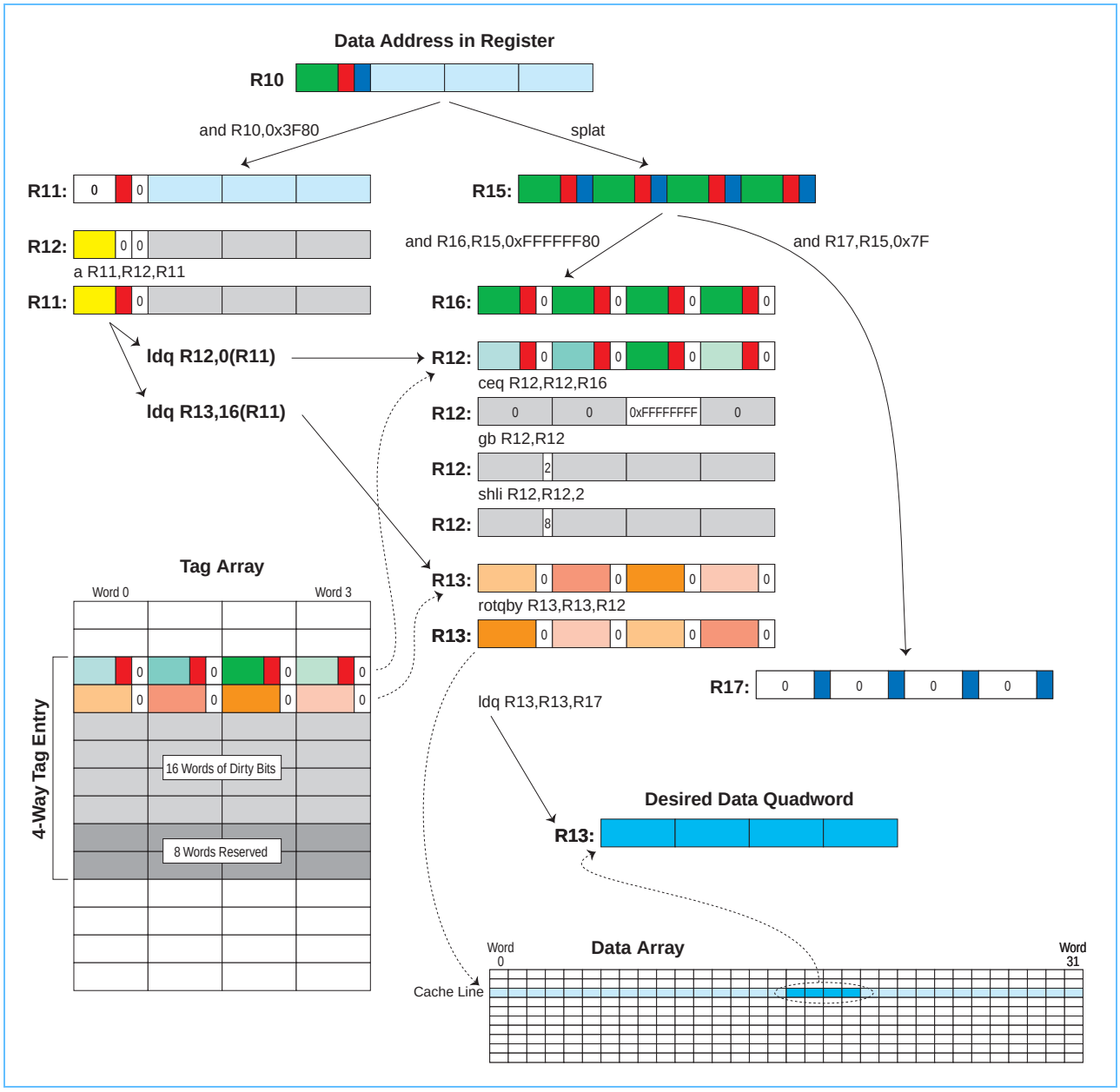
To support the software cache, a separate directory is maintained in each SPE. The compiler, using interprocedural analysis, replaces a subset of load and store instructions with instructions that explicitly look up the effective address of the data in a directory. If the directory lookup indicates that the data is present in the LS, the address of the requested variable cached in the LS is computed and the load or store proceeds using this LS location. Otherwise, a miss-handler

subroutine is invoked and the requested data is transferred from main storage. As is usual in hardware caches, the directory is updated and typically another line is selected for eviction, in order to make room for the new data.

Figure 21-1 on page 615 illustrates a software-cache lookup. Starting with the address of the desired variable in the left-most (or preferred) slot of an SPE register, instructions are executed to produce the address of the appropriate directory entry in the Tag Array. This task consists of masking out all bits in the address except for those that are used to index into the Tag Array. The cache-line size for the CBE processor is 128 bytes. By also making the directory entry 128 bytes long, the index obtained after masking does not need to be shifted right and multiplied, and it can be used directly as an offset into the Tag Array.

In the example, we show a 64KB 4-way cache, so there are also 128 lines in each way, and the index is thus also 7 bits. To this offset we add the base address of the Tag Array, and using this address we load two consecutive quadwords from the array. The first contains four tags, for the four ways of the cache, and the second contains the LS addresses of the four lines in the Data Array. Independent of this computation, the data address is replicated in all four slots of another register. This register is then masked twice to produce, in the first instance, four identical tags, and in the second, a register containing the offset in the line of the desired data. These two instruction sequences are independent and may be scheduled for parallel execution. We now do a SIMD compare of the loaded tags and the replicated tag from our data address. If any of the comparisons contains the tag we are seeking, then the 128-bit result of the compare will be non-zero, and we can use this to test for a hit. If the result is zero, we have a miss and the miss handler (not shown in the figure) is invoked.

Figure 21-1. Software-Cache Lookup



The call to the miss handler need not be expanded by the compiler until very late, so it does not appear as a call to the compiler optimizations. This allows the common case, which we assume to be a hit, to suffer no penalty due to the call setup, but it also means that the miss handler cannot use the system-register conventions. If there is a hit, as shown in *Figure 21-1*, we can use the result of the compare to determine the rotation needed to place the correct line address in the preferred slot, making it ready to be used as a LS address. Finally, we add the line offset to the line address, and we load the desired data. Store processing proceeds in much the same way,

but additional instructions are required to set the dirty bits. These bits, which are stored in the Tag Array entry, are used to ensure correct execution when multiprocessing ("correct execution" here means ensuring that updates are correctly reflected in main storage).

It takes roughly 12 instructions to process a cache hit, and a similar number to set dirty bits for a store, but since some of these instructions are dual-issued, and there is often other independent work that can be simultaneously scheduled, the cost in cycles is not high. Still, it is clearly important to attempt to reduce the number of cache lookups inserted when compiling a program.

This software cache can be used for either a serial or a parallel program. There is some additional cost involved in supporting parallel execution, since we must keep track of which bytes in a cache line have been modified in order to support multiple writers of the same line. If the compiler knows that no data will be shared between SPEs (for example, in the case of a serial SPE application), the additional cost of maintaining these dirty bits can be avoided.

Since the compiler must deal with the potential for aliases between cached data and data obtained by other means (for example by means of explicit prefetch in the compiler's tiler), the miss handler is required to take more care in choosing a line to be evicted than is normally the case with a hardware implementation.

21.5.5 Compiler and Runtime Support for Code Partitioning

Since the limited LS of each SPE must accommodate both code and data, there is always a possibility that a single SPE object will be too large to fit in the LS. Compilers can provide a code-partitioning technique to reduce the impact of the LS limitations on the program's code segment. This approach can be used standalone with an SPE compiler or by programmers choosing to manually partition their algorithms. When using OpenMP with single source compilation, the code partitioning can be integrated with the data software cache to allow for the execution of large functions with large data sets to run seamlessly across multiple SPEs.

In one approach to code partitioning, the compiler is used to automatically divide the SPE program into multiple partitions. The basic unit of partitioning is a function. Just as with data in the software-cache approach, the home locations of code partitions are in main storage. These SPE code partitions are overlaid during linking. That is, they are all assigned to the same starting virtual address. SPE code can then fit into a virtual-address space equal to the size of the largest code partition. Since the compiler determines how code is partitioned, it controls the partition size, and thus controls the amount of space used in the LS for code.

The overlaid partitions cannot execute at the same time. This implies that if code in one partition calls a function in another partition, the two partitions need to be swapped in and out of the LS at the point of the function call and return. To run a partitioned program, such partition transitions must be handled properly, and this can be done collaboratively by the compiler and a runtime partition manager.

When the compiler partitions SPE code, it can also reserve a small portion of the SPE LS for the runtime partition manager. The reserved memory can be divided into two segments: one to hold the continuously resident partition manager, and the other to hold the active code partition. The code partitions need to be relocatable, which implies that function calls should not use absolute addressing. The partition manager is responsible for loading partitions from their home location in main storage into LS during an interpartition function call or an interpartition return. The compiler

can modify the original SPE program to replace each interpartition call with a call to the partition manager. The partition manager also modifies the return address on the stack before branching to the called function to ensure that control returns to the partition manager first.

When an interpartition call is directed through the partition manager, a function pointer and arguments to the function are passed to the partition manager. The partition manager must determine which partition contains the called function. For this purpose, the compiler can assign an index to all partitions it creates, and it can encode the corresponding index in the function pointer that is passed back to the partition manager. An SPE pointer is 32-bit, but since the LS is 256 KB, only 18 bits are used. Of the 14 unused bits, 13 bits can be used for the partition index (the most-significant bit can be used to indicate special handling for calls to certain library functions).

For example, an interpartition call to function `foo` in partition 3 can be transformed from `foo (arg1, arg2, ...)` to `call_partition_manager(3<<18 | foo, arg1, arg2, ...)`. The partition manager fetches the correct partition using the partition index, and transfers control to the proper location within the partition by using the lower 18 bits of the function pointer.

21.5.6 Thread Library

Most CBE software systems will include a thread library, and many such libraries will follow the POSIX definition, documented as *POSIX threads explained* at http://www-128.ibm.com/developerworks/linux/library/l-posix3/index.html. A thread library can be used to create, manage, and terminate threads of execution, manage access to shared data using mutexes and condition variables, and distribute and manage workloads among threads.

Although programming directly with threads is powerful, it puts a burden on the programmer. You are responsible not only for protecting access to all shared data, but also partitioning and allotting chunks of work among threads. You need to explicitly manage a fork and join model or some other programming model. A program that explicitly uses threads can be a challenge to debug, unless you have access to a debugger that has been taught to track threads of execution.

Excellent books are available on this subject. *Programming with Threads*, by Steve Kleiman is a good place to start. *Threads Primer: A Guide to Multithreaded Programming*, by Bill Lewis is a gentle introduction. There are also many good web sites.

22. SIMD Programming

22.1 SIMD Basics

Both the PPE and the SPEs support parallel processing of Single Instruction Multiple Data (SIMD) vector elements. A *vector* is an instruction operand containing a set of data elements packed into a one-dimensional array, as shown in *Figure 22-1*. In the CBE processor, the vector elements can be fixed-point (integer) or floating-point values. Almost all Vector/SIMD Multimedia Extension and SPU instructions operate on vector operands. Vectors are also called *SIMD operands* or *packed operands*.

Figure 22-1. A Vector With Four Elements

As the SIMD name implies, this style of programming allows one instruction to be applied to the multiple data elements of a vector in parallel. In this way, SIMD processing exploits data-level parallelism. SIMD programming is prevalent in multimedia, graphics-intensive stream processing (such as gaming), and high performance computing—basically, in any compute-intensive application. The CBE processor is designed to operate efficiently on SIMD code, so programmers benefit from learning how to efficiently exploit data parallelism in their programs and how to take advantage of compiler optimizations for SIMD code.

Support for 128-bit-wide SIMD operations is pervasive in the CBE processor. In the PPE, these operations are supported by the 32-entry vector register file, Vector/SIMD Multimedia Extensions to the PowerPC instruction set, and C/C++ intrinsics for the Vector/SIMD Multimedia Extensions, as summarized in *Section 2* on page 45. In the SPEs, SIMD operations are supported by the 128-entry vector register file, SPU instruction set, and C/C++ intrinsics, as summarized in *Section 3* on page 63. In both the PPE and SPEs, the vector registers hold multiple data elements as a single vector. The data paths and registers supporting SIMD operations are 128 bits wide, corresponding to four full 32-bit words. This means that four 32-bit words can be loaded into a single register, and, for example, added to four other words in a different register in a single operation. Similar operations can be performed on vector operands containing 16 bytes, 8 halfwords, or (for the SPEs) 2 doublewords.

In a traditional vector processor, multiple vector elements (typically up to 64) are processed one after another, in a pipelined fashion. Registers are very wide—each one holding, for example, 64 vector elements of 64 bits each—but there are only a small number of such registers (for example, eight). These processors have elaborate memory instructions, such as *gather data* and *scatter data*, that use a vector register as an index into another array. For example, two vector loads can be used to fetch the values of `a[b[0...63]]`, as detailed below:

```
r1 = vector load b[0..63];
r2 = indirect vector load into array of a, using r1 as index
```

In contrast, a SIMD processor, such as an SPE, stores fewer vector elements in each register but it has far more of such registers. The memory unit of a SIMD processor is more similar to the memory unit of a scalar load-store processor than to the memory unit of a traditional vector processor. The most important distinction is that, in the newer SIMD processors, memory accesses must be aligned on 16-byte boundaries, and loads and stores operate only on 16 bytes at a time. Architectures that support only loads and stores that are aligned to the vector-register length, such as the CBE processor, are called architectures with *alignment constraints*. An *aligned reference* means that the desired data resides at an address that is a multiple of a vector register's 16-byte width.

This chapter describes some basic techniques of SIMD application programming and the reasons behind it—such as techniques used by compilers to extract parallelism from SIMD application programs. The material is intended for both application programmers and compiler writers. Some material is of particular interest to compiler writers. Nevertheless, an understanding of the compiler-related concepts is very useful for application programmers who wish to get the best performance from their programs.

22.1.1 Converting Scalar Data to SIMD Data

Depending on performance requirements and code size constraints, advantages can be gained by properly grouping the data that represent the elements of SIMD vectors.

When scalar code for an SPE is fed to a compiler, the compiler's backend code generator will typically expand a scalar load as read-aligned and a scalar store as a read-modify-write code sequence. This results in large code size with less than optimal performance. When the code is SIMDized (manually by the programmer and/or automatically by the compiler), such expansion of scalar code does not occur. The expansion of scalar code also does not occur in the case of the PPE, because the PPE has full-featured scalar units

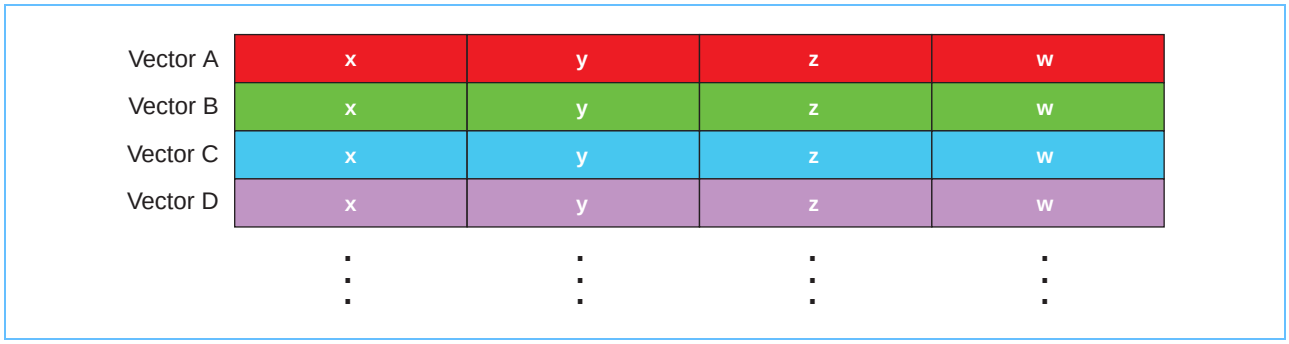
22.1.1.1 *Organizing Data*

Although there are several methods of organizing data, two forms are commonly used for SIMD data: an *array-of-structures (AOS)* form and a *structure-of-arrays (SOA)* form. Both are described in the sections below. For explanation purposes, 3-D homogeneous vertex coordinates will be used as the basic data type. Such a vertex consists of four floating-point components, x, y, z, and w.

Array Of Structures

The AOS form—also known as the *vector-across* or *vec-across* form—is shown in *Figure 22-2*. In this form, the vertex coordinates are structures, and several such structures represent an array of vertices. All components of a single vertex have the same color in *Figure 22-2*.

Figure 22-2. Array-of-Structures (AOS) Data Organization



In the AOS form, a set of vertices can be represented as an array of `structs`:

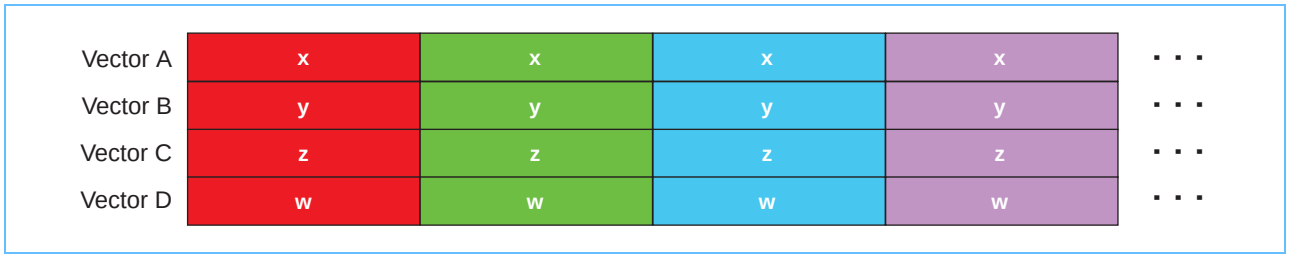
```
union {
    struct _coordinate{
        float x, y, z, w;
    } coordinate;
    vector float vertex;
} array[];
```

The `union` shown above specifies four single-precision floating-point values `x`, `y`, `z`, and `w`, which can be treated either as individual components, accessible one at a time, or as part of the packed vector, named `vertex`, accessible in a single vector operation.

Structure Of Arrays

The SOA form—also known as the *parallel-array* form—is shown in *Figure 22-3*. In this form, which is orthogonal to the AOS form, the components of the vertices are seperated into independent arrays. These arrays are arranged so that each vector contains a single component from four independent vertices. As in *Figure 22-2*, all elements of a single vertex have the same color in *Figure 22-3*.

Figure 22-3. Structure-of-Arrays (SOA) Data Organization



In the SOA form, a set of vertices can be represented as a `struct` of arrays:

```
struct {
    float x[];
    float y[];
    float z[];
```

```
    float w[];
} vertices;
```

If data is stored in the AOS form, the **spu_shuffle** intrinsic can be used to transform it to the SOA form, and vice versa.

Converting Between AOS and SOA Organization

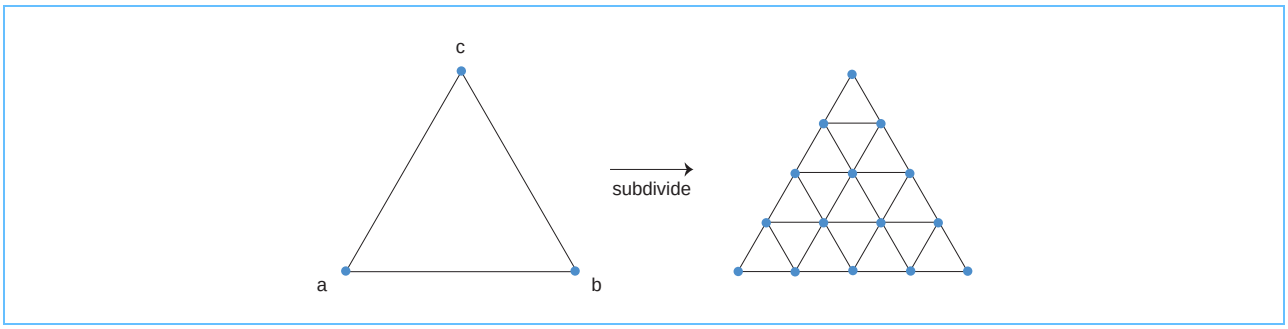
Converting between the AOS vectors (A, B, C, D) shown in *Figure 22-2* and the SOA vectors (A, B, C, D) shown in *Figure 22-3* entails performing a 4 x 4 matrix transpose. An efficient transpose can be accomplished using only two shuffle patterns, as follows:

```
vector unsigned char hi = ((vector unsigned char){
0x00,0x01,0x02,0x03,0x10,0x11,0x12,0x13,0x04,0x05,0x06,0x07,0x14,0x15,0x16,0x17});

vector unsigned char lo = ((vector unsigned char){
0x08,0x09,0x0A,0x0B,0x18,0x19,0x1A,0x1B,0x0C,0x0D,0x0E,0x0F,0x1C,0x1D,0x1E,0x1F});

tmp1 = spu_shuffle(A, C, hi);
tmp2 = spu_shuffle(A, C, lo);
tmp3 = spu_shuffle(B, D, hi);
tmp4 = spu_shuffle(B, D, lo);

A = spu_shuffle(tmp1, tmp3, hi);
B = spu_shuffle(tmp1, tmp3, lo);
C = spu_shuffle(tmp2, tmp4, hi);
D = spu_shuffle(tmp2, tmp4, lo);
```

22.1.1.2 *Triangle-Subdivision Example*

As an example of SIMDizing an algorithm, consider subdividing a single triangle into multiple smaller triangles, as shown in *Figure 22-4*. This is known as *triangle subdivision*. The original triangle is defined by the three vertices, `a`, `b` and `c`. Two levels of subdivision results in a subdivided triangle consisting of 16 smaller triangles defined by 15 vertices.

Figure 22-4. Triangle Subdivision



For this algorithm, there are several methods of exploiting SIMD parallelization in the generation of the subdivided vertices. Reviewing these methods should inspire programmers to consider alternate, non-traditional, methods that could result in more efficient solutions, in both development time and execution performance.

Method 1: Evaluate One Vertex at a Time Using AOS

In the AOS data-organization form (*Figure 22-2* on page 621), the coordinates for each vertex are packed into a vector. The three vertices that describe the original triangle in *Figure 22-4* can be represented as three vectors of four elements each. Each vertex is then evaluated one at a time. Performance can be improved by unrolling the loops and evaluate several vertices at a time. The additional instructions within the loop can be interleaved, manually or by the compiler, so that dependent-instruction latency can be hidden.

Although the AOS form is a natural fit for 3-D vertices exploiting SIMD methods, it is typically not the most efficient. Efficiency is compromised when the 3-D vertices have only three components. In addition, geometric operations such as dot products and cross products are not ideal. Dot products on the AOS form generate scalars, and cross products require vertex-component reordering.

Method 2: Evaluate One Vertex at a Time Simultaneously for Four Triangles Using SOA

Another way to subdivide the triangle is to evaluate one subdivision vertex at a time, over four independent triangles. In this approach, which uses the SOA data-organization form (*Figure 22-3* on page 621), each vector is populated with *independent* data, so a program can be developed as though the algorithm were operating on scalar data. Each vertex component—for example, the `x` component of each of four triangle vertices—is stored in a single vector.

Think of the data as if it were scalar, and the vectors were populated with independent data across the vector. This differs from the AOS form, in which the four values of each vertex are stored in one vector. *Figure 22-3* on page 621 shows the use of SIMD vectors to represent the four single-precision floating-point values `x`, `y`, `z`, and `w` for one vertex of four triangles. Not only are the data types the same across the vector, but now their data interpretation is the same. Depending on the algorithm, software might execute more efficiently with this SOA data organization than with the AOS organization shown in *Figure 22-2* on page 621.

Assuming that the algorithm can be independently and simultaneously performed on multiple data objects, this approach often is the most efficient.

Method 3: Evaluate Four Vertices at a Time Using SOA

A third approach is to evaluate four subdivision vertices for each triangle, one at a time. In this case, the vertex component data is replicated across the vectors. Unique weighting factors are maintained in each element of the vector, so that four subdivided vertices can be computed in parallel. However, inefficiency can result when the number of subdivision vertices is not a multiple of four. In the case of the two levels of subdivision shown in *Figure 22-4* on page 622, three iterations of four vertices and one iteration of three verticies would be performed.

22.1.2 Approaching SIMD Coding Methodically

Two of the principal objectives of SIMD programming are:

- Extract parallelism from both loops and basic blocks.
- Satisfy the constraints of data alignment[50], data-size conversions, and data stride[51].

These objectives can be achieved in both SPE code and PPE Vector/SIMD Multimedia Extension code.

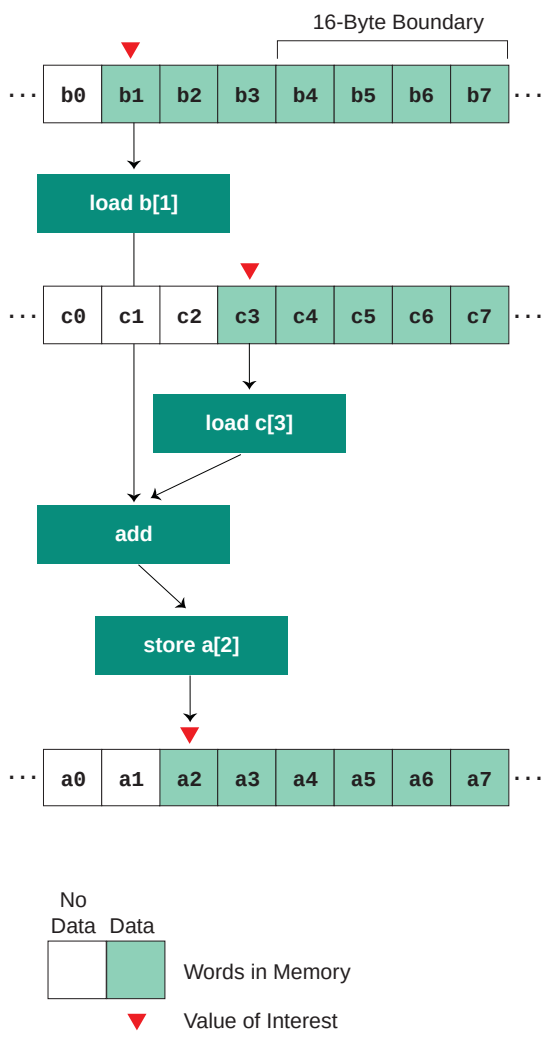
Consider the execution of the following loop:

```
for (i=0;i<65;i++) a[i+2] = b[i+1] + c[i+3]
```

This loop can be performed in a traditional sequential way as shown in *Figure 22-5*. Here, the first pair of 32-bit operands, `b[1]` and `c[3]`, are loaded from their respective addresses in memory using standard (non-vector) load instructions. Then, they are added together, and the result, `a[2]`, is stored using a standard (non-vector) store instruction. This produces a correct result, but the operands and the result are offset from one another, relative to the alignment of vectors of which they could be a part, so the method cannot be SIMDized without some realignment of the operands.

50. Alignment on 16-byte boundaries.
51. A *data stride* is a memory access pattern in which each element in a list is accessed sequentially.

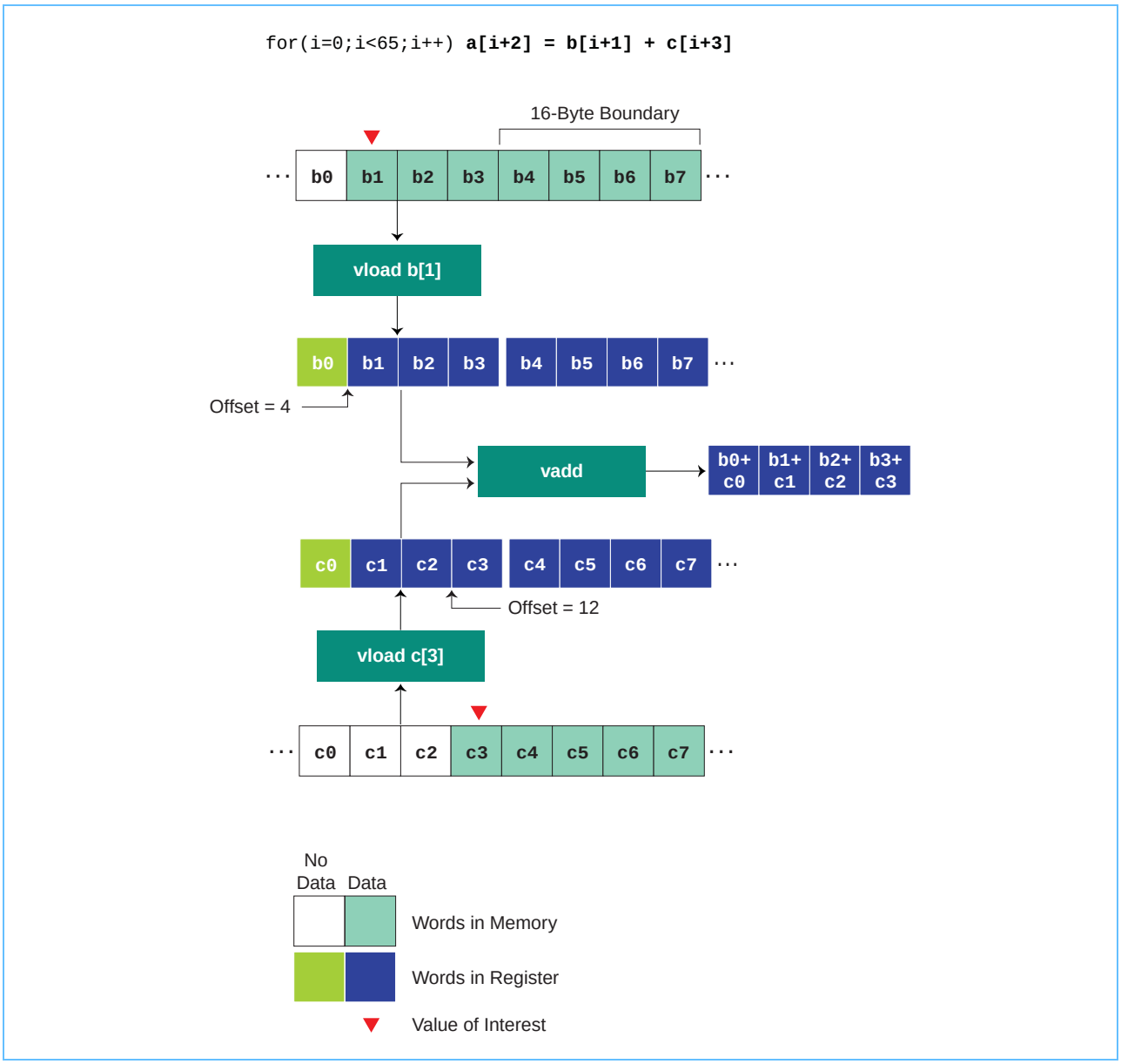
Figure 22-5. Traditional Sequential Implementation of a Loop



22.1.2.1 *Simple, Incorrect SIMDization*

A simple, but incorrect, SIMD implementation of the loop is illustrated in *Figure 22-6*. Here, vector-load instructions (called **vload** in this example) load the complete four-element vectors containing the `b[i]` and `c[i]` words. Then, a vector-add instruction sums the four pairs of vector elements in parallel. However, the result is not correct because the alignments of the `b[i]` and `c[i]` vector elements within the vector registers do not match—the `b[1]` element is added to `c[1]` rather than to `c[3]`.

Figure 22-6. Simple, Incorrect SIMD Implementation of a Loop

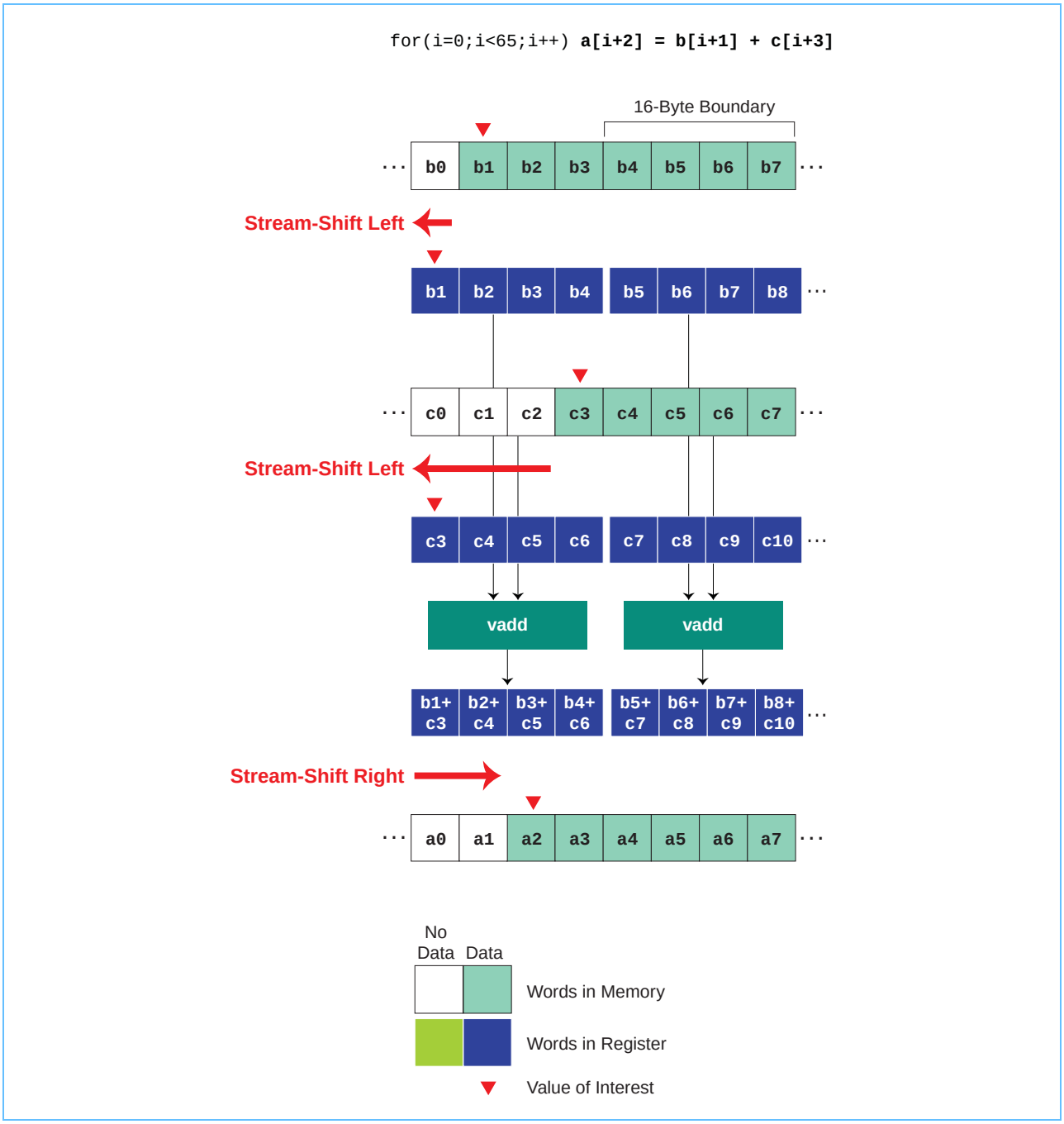


22.1.2.2 ***Simple, Correct SIMDization***

A simple, and correct, SIMD implementation of the loop is illustrated in *Figure 22-7*. It uses operations, called *stream-shift operations*, which shift memory loads or stores the right or left in order to align vectors within registers. A *stream* is a sequence of contiguous memory locations that are accessed by a memory reference throughout the lifetime of a loop (also called a *memory stream*), or a sequence of contiguous register values that are produced by an operation over the lifetime of a loop (also called a *register stream)*.

In the *Figure 22-7* example, each memory stream is shifted left, by different amounts, in order to place the vector elements, `b[1]` and `c[3]`, at corresponding register locations. A vector-add instruction sums the four pairs of elements in parallel, and these additions are repeated for subsequent vectors. A final stream-shift operation, to the right, is used prior storing the results so that the first result, `a[2]`, is stored in the expected memory location.

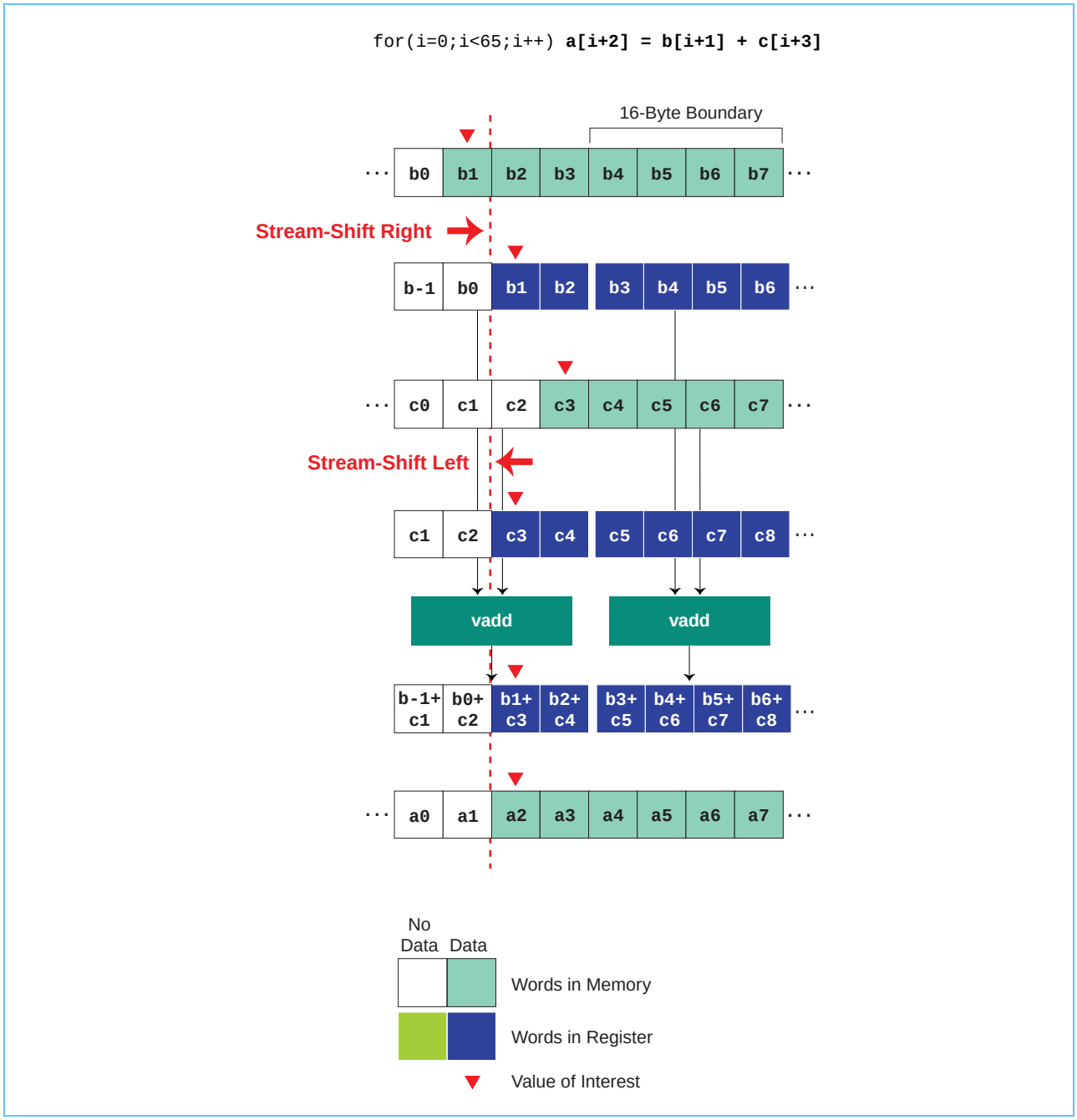
Figure 22-7. Simple, Correct SIMD Implementation of a Loop



22.1.2.3 *Optimized, Correct SIMDization*

An optimized, and correct SIMD implementation of the loop is illustrated in *Figure 22-8*. Here, the `b[1]` memory stream is shifted right and the `c[3]` memory stream is shifted left, so that both the `b[1]` and `c[3]` elements align at the same register position from which the result can be stored back to memory without stream-shifting. The advantage of this method, compared with that in *Figure 22-7*, is the elimination of the last stream-shift.

Figure 22-8. Optimized, Correct SIMD Implementation of a Loop



22.1.2.4 *SIMDization Code Structure for a Loop*

The basic code structure for the SIMDization of a loop is:

```
<SIMDization prolog>
for(i=0; i<65; i+=4)
   <SIMDization loop body>
<SIMDization epilog>
```

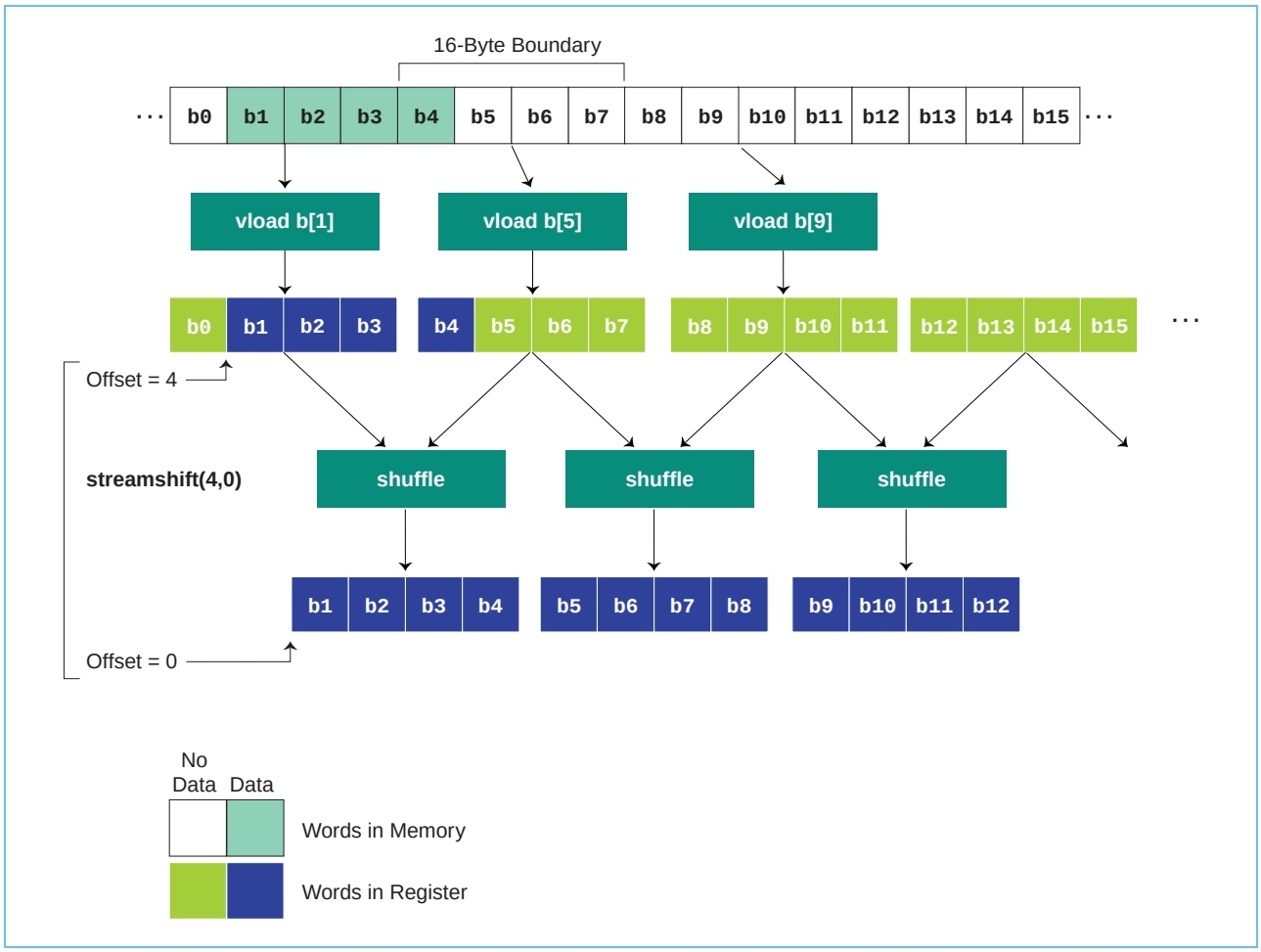
The *SIMDization prolog* (*Figure 22-10* on page 632) is used if the data to be stored is misaligned. The *SIMDization epilog* (*Figure 22-11* on page 633) is used, depending on the requirements for store alignment and the loop's trip-count.

22.1.2.5 *Implementing Stream-Shift Operations on Data*

Figure 22-9 shows how to implement stream-shift operations using shuffle operations (the **shufb** instruction for an SPE, or the **vperm** instruction for the PPE). The operations on `b[i]` require a stream of vectors, starting at memory address `b[1]`, in vector registers. To obtain this, the vector that contains `b[1]`—the vector that begins at address `b[0]` in memory—is first loaded into a vector register with a vector-load instruction, followed by similar loads of subsequent vectors. After the loads are complete, shuffle operations are performed on the contents of the registers so that the `b[1]` element in the first vector is shifted from register offset 4 to offset 0. This operation is denoted in the figure as `streamshift(4,0)`, indicating that offset 4 is moved to offset 0. The pattern for either the **vperm** or **shufb** operation for `streamshift(4,0)` is: (0x'04050607', 0x'08090A0B', 0x'0C0D0E0F', 0x'10111213').

Similar stream-shift operations are performed on the vector stream containing `c[3]`. Then, the adds and a store are performed, as shown in *Figure 22-8* on page 629.

Figure 22-9. Implementing Stream-Shift Operations for a Loop

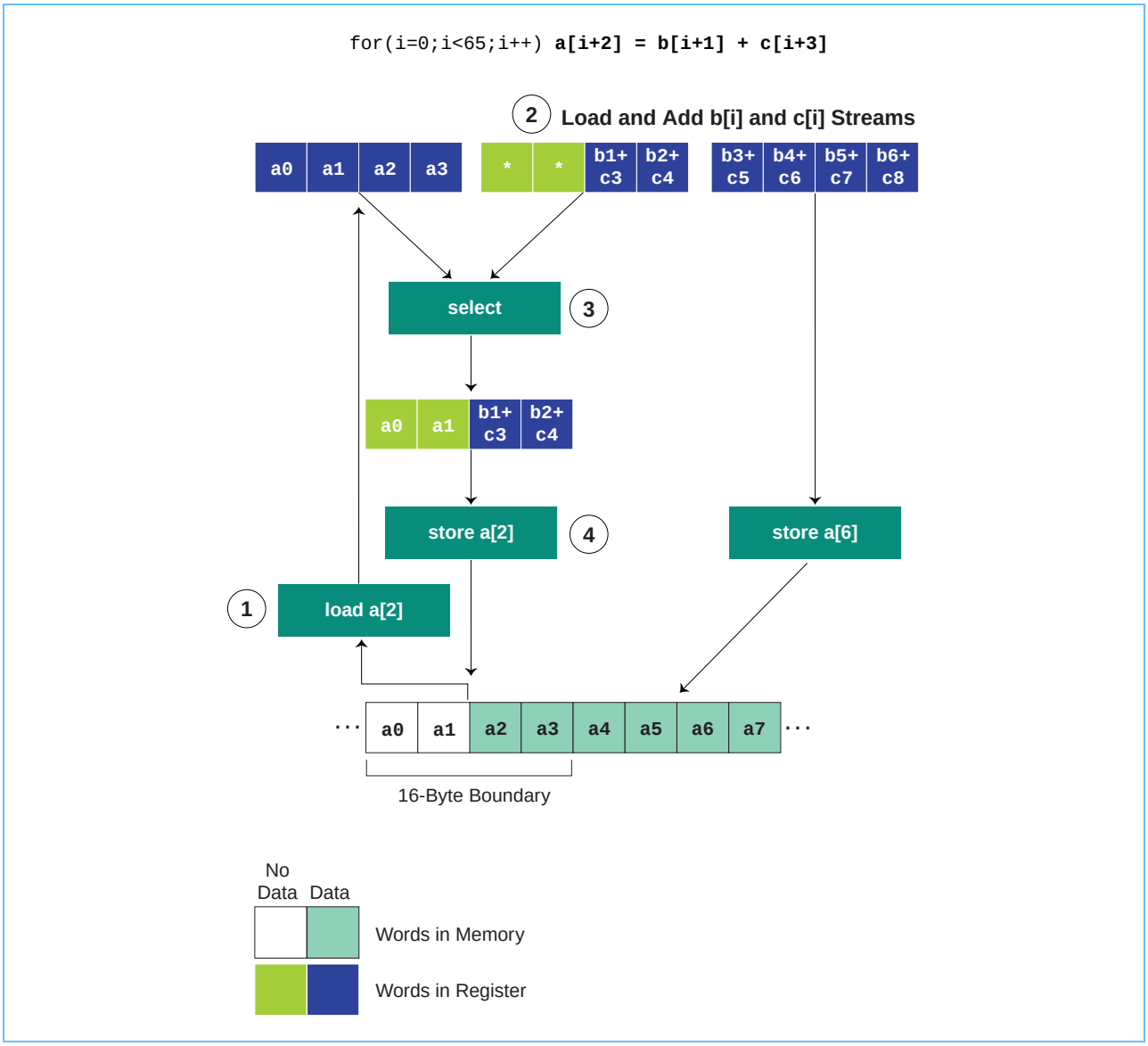


22.1.2.6 *The Prolog*

Figure 22-10 shows how to implement a SIMDization prolog, which is used if the data to be stored is misaligned:

1. The vector that contains `a[2]`—the vector that begins at address `a[0]` in memory—is loaded.
2. The `b[i]` and `c[i]` streams are loaded and added as in *Figure 22-8* on page 629.
3. The most-significant two elements of the `a[i]` vector and the least-significant two elements of the first add result are selected.
4. The result of the select is stored, starting at the memory address of `a[0]`.

Figure 22-10. SIMDization Prolog

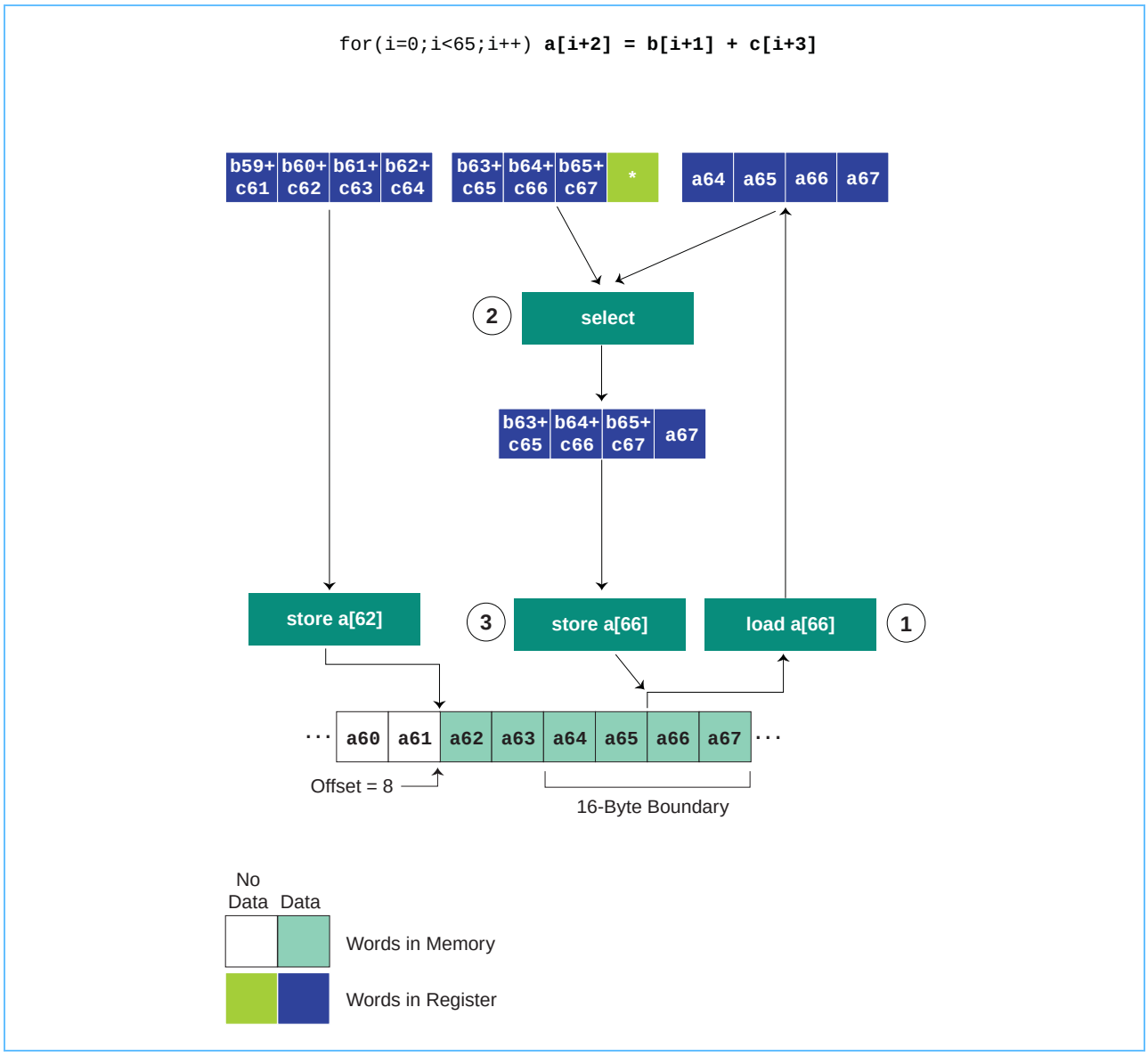


22.1.2.7 *The Epilog*

Figure 22-11 shows how to implement a SIMDization epilog, which depends on the requirements for store alignment and the loop's trip-count. In this example, the trip count is 66. Thus, the last-computed vector result has only three of its four elements with valid data. To handle this:

1. The `a[i]` vector containing the last element, `a[66]`, is loaded.
2. This `a[i]` vector and the three valid result elements are selected.
3. The vector containing the `a[66]` result is stored.

Figure 22-11. SIMDization Epilog



22.1.2.8 *Machine-Independent Pseudocode*

The machine-independent pseudocode result of the above SIMDization operations—before software pipelining of adjacent loads—is shown below. The code assumes that:

- All operations are on vectors.
- Loads and stores are normalized to truncated addresses.
- For SPE application, the **splice**, **shiftpairl**, and **shiftpairr** operations are data-reorganization operations that map to the **spu_shuffle** or **spu_sel** intrinsics.

```
i = 0;
a[i] = splice(a[i], shiftpairr(b[i - 4], b[i], 4) + shiftpairl(c[i - 4], c[i],4), 8);
i=4;
do {
   a[i] = shiftpairr(b[i-4], b[i], 4) + shiftpairl(c[i-4], c[i],4);
   i = i + 4;
} while (i < 65-4);
a[i] = splice(shiftpairr(b[i-4], b[i],4) + shiftpairl(c[i-4], c[i], 4), a[i],12));
```

22.1.2.9 *Typical SPE Code*

A typical example of actual code for an SPE—after software pipelining of loads, loop normalization, and address truncation—is shown below.

```
extern vector unsigned int a[], b[], c[];            // quadword aligned

int i;
vector unsigned int tc, tb, oldSPCopy0, oldSPCopy1;

vector unsigned char patt_b = (vector unsigned char){12,13,14,15,16,17,18,19,
                                                      20,21,22,23,24,25,26,27};
vector unsigned char patt_c = (vector unsigned char){4,5,6,7,8,9,10,11,
                                                      12,13,14,15,16,17,18,19};

tc = c[1];
tb = b[0];

a[0] = spu_sel(a[0],spu_add(spu_shuffle(b[-1],tb,patt_b),
             spu_shuffle(c[0], tc,patt_c)), (vector unsigned int)spu_maskb(255));

i = 0;
do {
   oldSPCopy0 = tc;
   oldSPCopy1 = tb;

   tc = c[i+2];
   tb = b[i+1];
```

```
   a[i+1] = spu_add(spu_shuffle(oldSPCopy1,tb,patt_b),
                    spu_shuffle(oldSPCopy0,tc,patt_c));

   i = i + 1;
} while (i < 15);

a[16] = spu_sel(spu_add(spu_shuffle(tb,b[16],patt_b),
           spu_shuffle(tc,c[17],patt_c)), a[16],(vector unsigned int)spu_maskb(15));
```

22.1.3 Coding for Effective Auto-SIMDization

Before going further into the subject of SIMD programming, here are some general application-programming guidelines for writing code that can be effectively SIMDized by a compiler—a method called *auto-SIMDization*: Details about why these methods work well are discussed in subsequent sections.

22.1.3.1 *Organize Loops*

Organize loops so that they can be auto-SIMDized. This may not be easy, but it is possible. Such loops should be inner-most loops. They should use the `for` loop construct, not the `while` loop construct; the `while` loop construct is not SIMDizable.

Constant Trip Count

A constant trip count[52] is best. But in any case, the trip count should be more than three times the number of elements per vector. Loops with long trip count results in more efficient SIMDization. Generally, a trip count of more than three times the number of elements per vector is sufficient for SIMDization. But shorter loops might also be SIMDizable, depending on their alignment and/or the number of statements in the loop. For example, a loop such as:

```
for(i=0; i<4;i++) a[i] = xxx;
```

The above loop has a trip count of four and an array, `a[i]`, of `int`. It can be auto-SIMDized if `a[0]` is aligned.

The reason that the trip count should be more than three times the number of elements per vector is because there must be at least one iteration in the loop for auto-SIMDization to work. Since the prolog and epilog could each require one iteration, three iterations of the loop may be required if the loop has more than two statements and the compiler knows nothing about the alignment.

Declare Constant Loop Bounds

Use `#define` to declare constant loop bounds. Doing so increases the reliability that the compiler will use the correct constant.

52. “Trip count” is the number of iterations in a loop.

Use the Select Operation for Conditional Branches

Loops that contain if-then-else statements may not always be SIMDizable. However, if the if-then-else statement has a comon left-hand variable and can be expressed using the C-language `:?` (colon question-mark) operator, the code will SIMDize using the *select bits* instruction. Here is an example of a realizable data-parallel select operation:

```
for (i = 0; i < VL; i++)
    if (a[i] > b[i])
        m[i] = a[i]*2;
    else
        m[i] = b[i]*3;
```

22.1.3.2 *Organize Data in Memory*

Lay out data in memory so that operations on it can be easily SIMDized.

- Use stride-one accesses. Non-stride-one accesses[53] are less efficiently SIMDized, if at all. Random or indirect accesses are not SIMDizable.
- Lay out the data to maximize aligned accesses.
- Use arrays; don't do your own pointer arithmetic in the application to access large data structures. Use global arrays that are statically declared. Arrays are aligned to 16-byte boundaries.
- If it is not possible to use aligned data, use the `alignx` directive to indicate to the compiler what the alignment is. For example:

  ```
  #pragma alignx(16, p[i+n+1]);
  ```

- If you know arrays are disjoint, use the disjoint directive to indicate to the compiler that the arrays specified by the pragma are not overlapping:

  ```
  #pragma disjoint (*ptr_a, b)
  #pragma disjoint (*ptr_b, a)
  ```

22.1.3.3 *Organize Algorithms*

Structure algorithms to reduce dependences:

- Loops with inherent dependences are not SIMDizable.
- Avoid using pointers when not absolutely necessary. Potential aliasing can confuse the compiler.

22.1.3.4 *Use Appropriate Computations*

Avoid computations that are not SIMDizable, including:

53. A stride of one refers to a memory access pattern in which each element in a list is accessed sequentially.

- Function calls are not SIMDizable. Use inline functions or macros in place of function calls. If function calls are kept, be sure to enable inlining by the compiler and possibly add an inlining directive to make sure that it happens.
- Operations that do not map onto native vector instructions are less SIMDizable. In general, all operations except load, branch, and hint-for branch are capable of being mapped.

22.2 Auto-SIMDizing Compilers

Compilers that optimize for traditional vector processors are called *vectorizing compilers*. To make the distinction between traditional vector processing and SIMD processing, we use the terms *SIMDization* and *SIMDize* to refer to vectorization on a SIMD processor.

A compiler that automatically merges scalar data into a parallel-packed SIMD data structure is called an *auto-vectorizing* compiler. Such compilers must handle all the high-level language constructs and therefore do not always produce optimal code. Although programming with intrinsics gives programmers a direct and powerful way to access the SIMD parallelization of SPU operations, this kind of programming can be challenging and time-consuming. In such cases, use of an auto-vectorizing compiler is generally less tedious and more productive than hand-coding, except perhaps for very critical kernel loops. But even in the latter cases, an auto-vectorizing compiler can often achieve performance that is nearly as good as hand-coding with intrinsics.

Although this section, and the remainder of this chapter, are intended primarily for compiler writers, application programmers can benefit from understanding the challenges and limitations facing a SIMDizing compiler, so that their application programs can avoid coding styles that are not readily SIMDizable by a compiler.

To understand the context of this discussion, consider the following example of SIMDizing a loop. The original, scalar loop multiplies two arrays, term by term. In the SIMDized version, the arrays are assumed to remain scalar outside of the subroutine, `vmult`.

```
/* Scalar version */
int mult(float *array1, float *array2, float *out, int arraySize) {
    int i;
    for (i = 0; i < arraySize; i++) {
        out[i] = array1[i] * array2[i];
    }
    return 0;
}


/* SIMDized version for an SPE */
int vmult(float *array1, float *array2, float *out, int arraySize) {

    /* This code assumes that the arrays are quadword-aligned. */
    /* This code assumes that the arraySize is divisible by 4. */

    int i;
    vector float *varray1 = (vector float *) (array1);
    vector float *varray2 = (vector float *) (array2);
    vector float *vout = (vector float *) (out);
```

```
    for (i = 0; i < arraySize/4; i++) {
        /* spu_mul is an intrinsic that multiplies vectors */
        vout[i] = spu_mul(varray1[i], varray2[i]);
    }

    return 0;
}
```

22.2.1 Motivation and Challenges

Compiling for the SPU's SIMD operations most closely resembles the work done with traditional loop-based vectorization, pioneered for vector supercomputers. However, architectural constraints for accessing memory and performing vector operations present new challenges.

22.2.1.1 *Misaligned Data*

One of the most significant challenges of SIMDization is dealing with misaligned data. Since the SPU hardware supports loads and stores only on quadword (16-byte) alignment, SIMDizing loops that operate on misaligned data accesses leads to incorrect results. Traditionally, a valid vectorization is constrained only by dependences: whether or not data in the loop depends on other data being computed first. SIMDization must satisfy three additional alignment conditions:

- The alignment of data in memory dictates the byte-offset in its destination vector register when it is loaded.
- Vector operations must operate on data that are at the same byte-offset in the input vector registers.
- Data being stored must be at the same byte-offset in the vector register as the memory alignment of the store address.

22.2.1.2 *Scatter-Gather*

Scatter-gather refers to a technique for operating on sparse data, using an index vector. A *gather* operation takes an *index vector* and loads the data that resides at a base address added to the offsets in the index vector. A *scatter* operation stores data back to memory, using the same index vector.

The CBE processor's SIMD architecture does not directly support scatter-gather in hardware. Some processors do support scatter-gather directly in hardware, but this capability comes at a great cost to the memory subsystem. Because the CBE processor does not support scatter-gather in hardware, the best way to extract SIMD parallelism is to combine operations on data in adjacent memory addresses into vector operations.

The CBE processor supports only stride-one loads and stores. Only data stored consecutively in memory is loaded or stored. Array accesses such as those represented by `a[2i+1]`, where `i` is an iteration index, result in poor memory-access patterns. Random indirect accesses are also a potential performance problem.

The CBE processor's architectural approach has the advantage of providing very efficient support for the most common types of memory access, while still providing relatively inexpensive support for rarer types of memory access, such as scatter-gather. This architectural approach enables the CBE processor to operate at faster clock speeds and higher compute density than would be the case if it supported scatter-gather in hardware.

22.2.1.3 *Packed Data*

Because SIMD vectors consist of packed data elements, a compiler writer must keep track of the number of data elements in each register. A packed vector refers to the fact that a 16-byte register may hold 16 1-byte values, eight 2-byte values, four 4-byte values, or two 8-byte values—for example, `char`, `short`, `int`, or `long long`, signed or unsigned. The number of values in a register changes when data is converted from one data type to another, such as when a vector of 1-byte `char` values is converted to a vector of 2-byte `short` values. In this case, the original vector expands to become two vectors.

22.2.1.4 *Mix of Data Types*

The mix of data types in multimedia programs can present problems. Traditional vectorizable programs contain loops with a single data type. In contrast, multimedia programs contain a rich mix of data types, and converting from one type to another is common.

Data is often stored as 1-byte `char`, but operated on as 2-byte `short`. Without taking care, it is easy to use the whole computation bandwidth by operating on eight `short` values at a time, but waste half the memory bandwidth by only storing eight `char` values at a time. A compiler writer must efficiently handle mixed data types in a loop to maximize both computation and memory bandwidth.

22.2.1.5 *Non-Uniform Instruction Set*

Not all data types are equally supported by the SPU instruction set. For example, vectors of 16-bit integers are the best supported. There is weak support for `vector long long` (vectors of 64-bit integers). There is slow throughput for `vector double` (vectors of double-precision floating-point numbers). 16-bit vector multiply is supported, but 32-bit vector multiply is not. There is no support for complex-number arithmetic.

22.2.1.6 *Parallelism Program Scope*

Although long-running loops were traditionally targets for optimization on vector processors, the relatively short vectors in SIMD architectures make SIMD programming attractive even for short, unstructured computations. In many multimedia programs, SIMD parallelism can be extracted from computations among the x, y, and z coordinates of a vertex. Multimedia programs often contain manually unrolled loops and loops with a short trip count. A compiler writer must be able to extract SIMD parallelism from various program scopes, such as a single basic block, an inner-most loop, or a loop nest.

The CBE processor provides plenty of support for complex-number arithmetic, such as the ability to perform SIMD and shuffle/permute operations. Most of the CBE processor's hardware support has been abstracted away from special operations (such as complex-number arithmetic) to

general operations on SIMD code—(for example, from `chars` to `double`). Thus, although there is no dedicated hardware for complex-number arithmetic, this results is higher overall performance for most SIMD applications.

22.2.1.7 *SIMDization Interaction with Other Optimizations*

SIMD functional units are more constrained than scalar functional units, SIMD vectors are relatively short, and multimedia programs are complex applications. Because of these considerations, the interaction between SIMDization and other compiler optimizations can be quite involved. For example, there is a trade-off between SIMDization and instruction-level parallelism (ILP). Without taking care, a compiler can SIMDize a loop but degrade performance.

Addressing each of these challenges is difficult in itself, but they can and do appear in actual applications all at once. The following code fragment exhibits several of these problems: misaligned access, length conversion, short loops and available SIMD parallelism across nested loops:

```
short input[], coef[];
for (i=0; i < NInputs; i++) {
    int sum = 0;
    for (j=0; j < 16; j++)
        sum += input[k+j] * coef[j];
    output[i] = sum;
}
```

22.2.2 Examples of Invalid and Valid SIMDization

Section 22.1.2 Approaching SIMD Coding Methodically on page 624 presents an example of invalid and valid SIMDization methods for a loop. This section presents a similar example but with a bit more detail.

Consider the following code fragment, in which integer arrays `a`, `b`, and `c` are quadword-aligned:

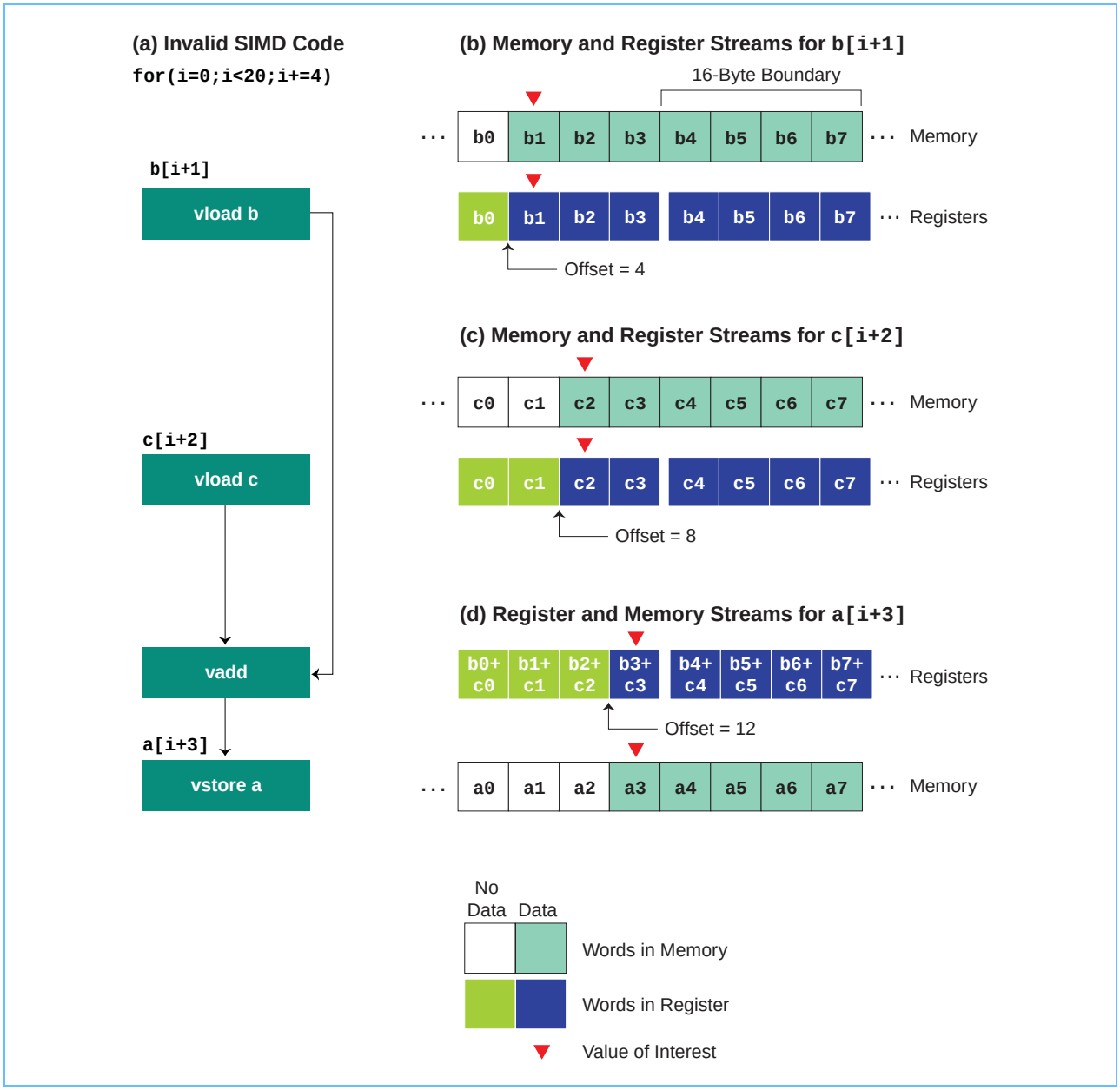
```
for (i = 0; i < 20; i++) {
       a[i+3] = b[i+1] + c[i+2];
}
```

Since there is no loop-carried dependence, the loop can be easily SIMDized for traditional vector processors that have no alignment constraints. However, such SIMDized code is invalid for SIMD units, such as the CBE processor, that support only aligned loads and stores.

Figure 22-12 shows the incorrect execution of this loop when it is SIMDized for traditional vector processors on hardware with alignment constraints. Consider the first iteration of the SIMDized loop in *Figure 22-12(a)*, especially the values of the expression `a[3]=b[1]+c[2]` that are highlighted with red triangles in items (b) through (d). The `vload b[1]` operation loads vector `b[0],...,b[3]` with b[1] at byte-offset 4 in its vector register. The `vload` operation is a compiler-internal function, described in *Section 22.3.4.1 Streams* on page 648. The operation `vload(addr(i))` loads a vector from a stride-one memory reference described by `addr(i)`. This produces a register stream whose byte-offset is the alignment of `addr(i)`.

Similarly, the `vload c[2]` operation loads `c[0],...,c[3]` with `c[2]` at byte offset 8, as shown in *Figure 22-12* (c). Adding these two vector registers yields the values `b[0]+c[0], ..., b[3]+c[3]`, as illustrated in *Figure 22-12* (d). This is not the result specified by the original `b[i+1]+c[i+2]` computation.

Figure 22-12. Invalid SIMDization on Hardware with Alignment Constraints



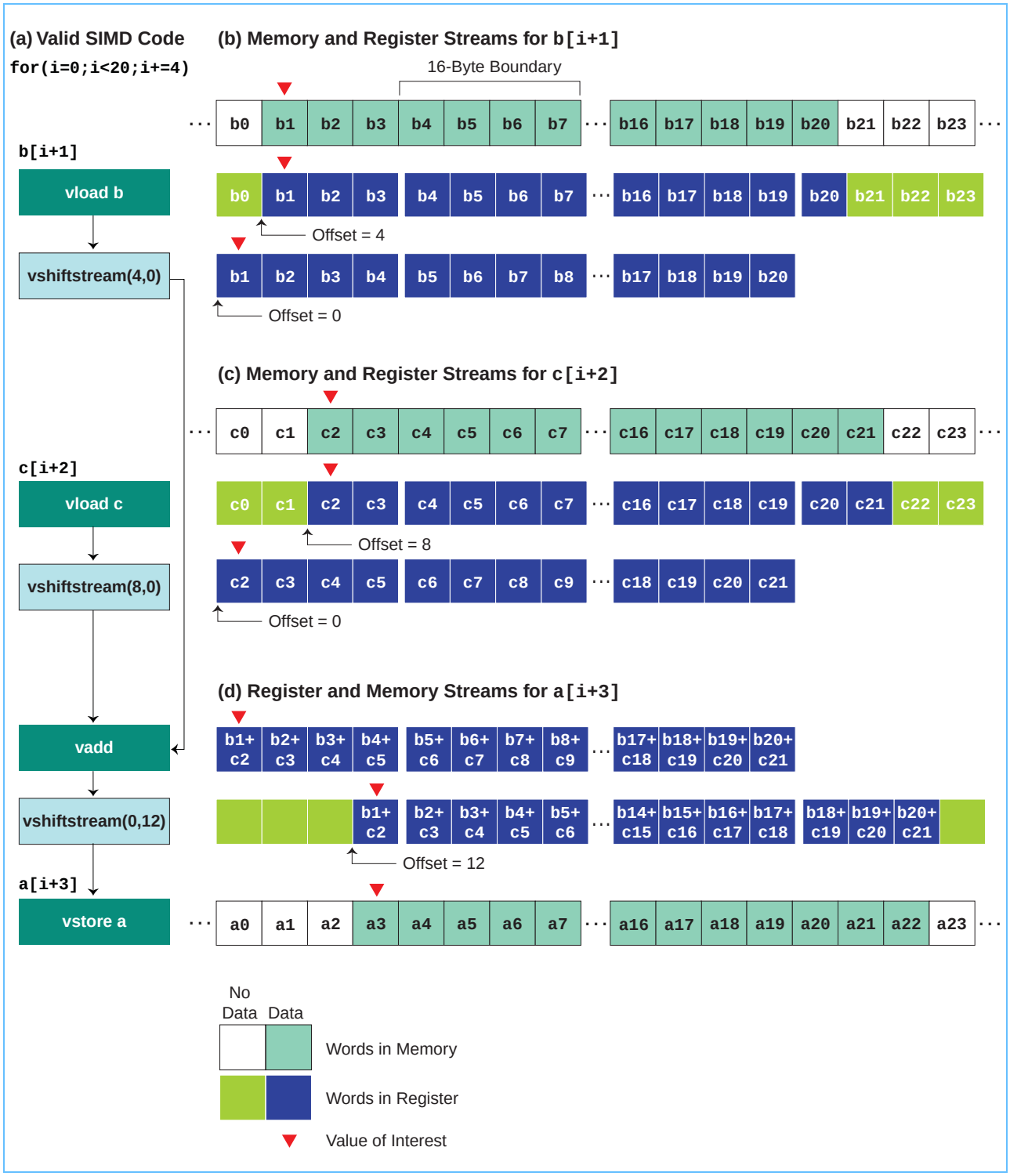
A valid vectorization for traditional vector units is only constrained by dependences. This is no longer sufficient when SIMDizing for architectures with alignment constraints. A valid SIMDization must satisfy these conditions:

- The alignment of data in memory dictates the byte-offset in its destination vector register when it is loaded.

- Vector operations must operate on data that are at the same byte-offset in the input vector registers.
- Data being stored must be at the same byte-offset in the vector register as in the memory alignment of the store address.

Figure 22-13 on page 643 shows one example of a valid SIMDization for the loop with misaligned references. A compiler can automatically generate data reorganization instructions to align data in registers to satisfy these alignment constraints. Leading and trailing **vstore** operations require store Prolog and Epilog, as described in sections *Section 22.1.2.6* on page 632 and *Section 22.1.2.7* on page 633.

Figure 22-13. An Example of a Valid SIMDization on SIMD Unit with Alignment Constraints



22.3 SIMDization Framework for a Compiler

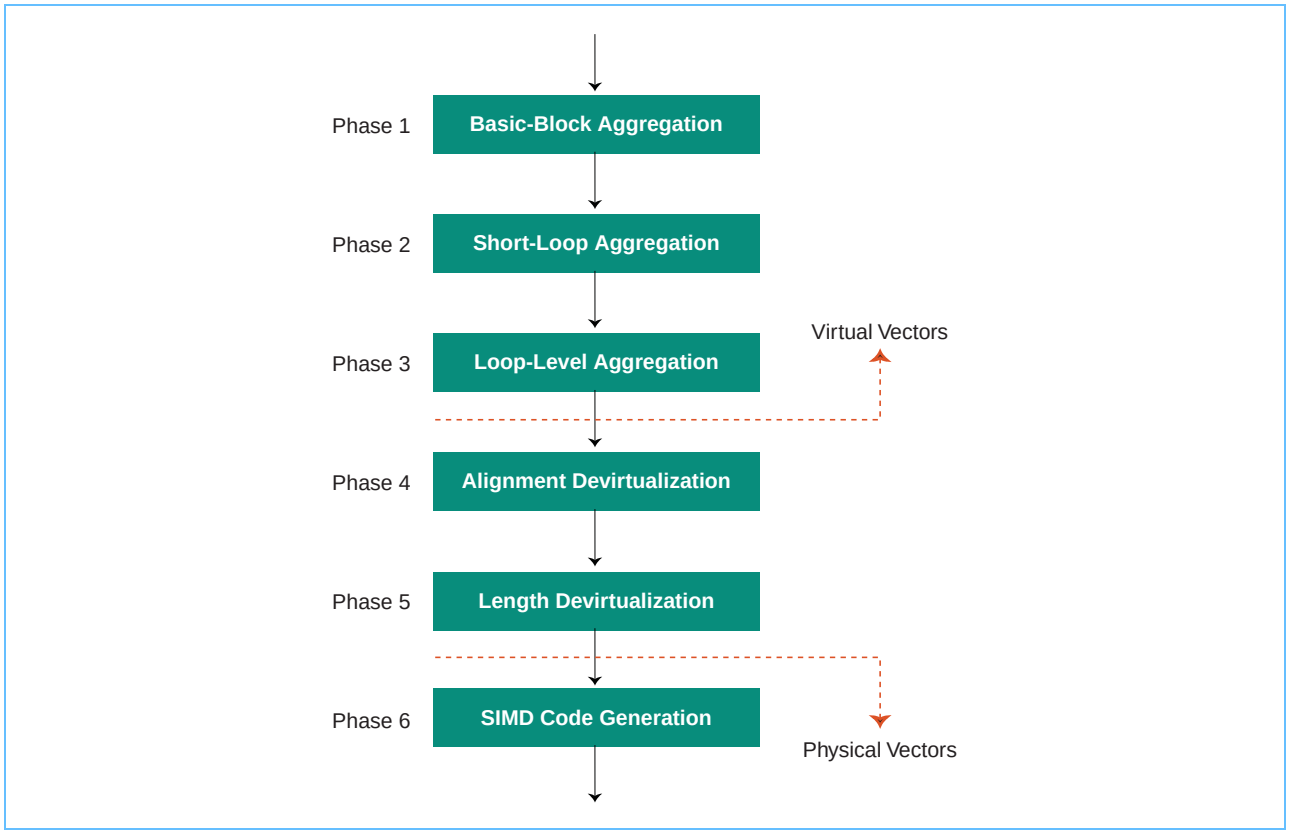
This section describes how a compiler can provide a SIMDization framework using a set of SIMDization algorithms. Although the intended audience is primarily compiler writers, the description may also be interesting to application programmers. It explains techniques for efficiently handling alignment and multiple-sourcing of SIMD data, and the conversion techniques that can be applied.

A SIMDization framework is presented that SIMDizes loops primarily with stride-one memory accesses. A stride of one refers to a memory access pattern in which each element in a list is accessed sequentially. However, the SIMDization framework we describe accomplishes more than SIMDizing the obvious candidates. Some of the features of the framework include:

- SIMD parallelism can be extracted from different program scopes, including basic block, inner-most loops, and inner-most loop nests.
- SIMDization can result in mixed scalar and SIMD code.
- Loops may contain multiple statements.
- Data inside the loops may have different data types.
- Loops may convert data from one type to another.
- Data inside the loops may have mixed lengths.
- Loops that are manually unrolled are SIMDized.
- Operations across vectors (reductions) can be SIMDized.
- Loops may contain induction, reduction, and private variables.
- Both compile-time alignment and runtime alignment can be performed.
- Interprocedural alignment analysis is used to obtain accurate alignment information.
- Programmer-inserted directives describe alignment of data (for example, the `_align_hint` intrinsic specified by the *SPU C/C++ Language Extensions* specification).
- Redundant sign-extensions are minimized, to reduce unnecessary packing and unpacking overhead.
- Programmer-inserted directives indicate which loops can be parallelized.
- Diagnostic information is provided on SIMDization failure.

The approach uses *virtual vectors* to carry out the early SIMD transformations. These vectors have no alignment constraints and can have any length. Compilers gradually devirtualize virtual vectors to physical vectors by later SIMD transformations. At that stage, architectural constraints, such as alignment and actual vector length, are taken into account. *Figure 22-14* on page 645 shows the six phases of the SIMDization process.

Figure 22-14. SIMDization Framework Based on Virtual Vectors



SIMDization is broken into six phases. Phases 1, 2, and 3 operate on virtual vectors (Basic-Block Aggregation, Short-Loop Aggregation, Loop-Level Aggregation). Phases 4 and 5 transition the virtual vectors to physical vectors (Alignment and Length Devirtualization). Phase 6 operates on physical vectors (SIMD Instruction Scheduling and Code Generation). The first three aggregation phases combine operations on data at adjacent memory into vector operations. Such contiguous memory access patterns are found in two kinds of places:

- At the basic-block level, adjacent memory accesses like `a.y` and `a.z`, or `b[i]` and `b[i+1]`, can be combined.
- At the loop level, stride-one accesses like `c[i]` across consecutive loop iterations can be combined.

Unfortunately, these two types of accesses are often not compatible. For example, although `d[i].x` and `d[i].y` are adjacent accesses at the basic-block level, they are not stride-one accesses at the loop level.

22.3.1 Phase 1: Basic-Block Aggregation

Basic-block aggregation extracts SIMD parallelism within a basic block by packing isomorphic computation on adjacent memory accesses into vector operations. That is, when data is being operated on in the same way, but the data is not already in a vector, the data is moved into a vector so that it can later be SIMDized. This is possible only if the data is stored in adjacent addresses in memory.

Such parallelism is often found in unrolled loops, whether the loops are unrolled manually by the programmer or automatically by the compiler. It is also common in graphic applications that manipulate adjacent fields of aggregates, such as the subdivision computation shown here:

```
for (i=0; i < n; i++) {
    t = triangles[i];
    v[i].x=w0 * t.p[0].x + w1 * t.p[1].x + w2 * t.p[2].x;
    v[i].y=w0 * t.p[0].x + w1 * t.p[1].x + w2 * t.p[2].y;
    v[i].z=w0 * t.p[0].x + w1 * t.p[1].x + w2 * t.p[2].z;
    v[i].w=w0 * t.p[0].x + w1 * t.p[1].x + w2 * t.p[2].w;
}
```

In this example, the last three computations can be packed into one vector statement. This is because they are isomorphic, and also operate on adjacent fields in memory (`x, y, z, w`). The first statement in this loop (`t = triangles[i]`) is an aggregate copy that can be SIMDized into a vector copy.

A loop SIMDized at the basic-block level can be further SIMDized at the loop-level. For example, if `x, y, z` and `w` are 16-bit integers, both basic-block and loop-level transformations are needed to extract enough SIMD parallelism for a 16-byte vector.

22.3.2 Phase 2: Short-Loop Aggregation

Short-loop aggregation eliminates inner loops that have short trip counts. SIMDizable short loops are collapsed into vector operations. The next loop in the original loop nest becomes the new inner-most loop. This expands the scope of subsequent SIMDization transformations.

For example, consider a simplified loop nest, which negatively indexes the array, `dp`:

```
for (j=40; j <= 120; j++) {
    sum = 0;
    for (i=0; i < 40; i++) {
        sum += wt[i] * dp[i-j];
    }
    L_result[j] = sum;
}
```

After short-loop aggregation, the inner loop is SIMDized into a single statement operating on a vector of 40 elements, since its trip count is 40. See *Section 22.3.4.1 Streams* on page 648 for a definition of `reduct`.

```
for (j=40; j <= 120; j++) {
    sum = 0;
    sum = sum + reduct(+, wt[0:39] * dp[-j:39-j]);
    L_result[j] = sum;
}
```

22.3.3 Phase 3: Loop-Level Aggregation

Loop-level aggregation extracts SIMD parallelism across loop iterations. Computations on stride-one accesses across loop iterations are combined into vector operations. The loop is blocked by `B`, defined such that the byte length of each virtual vector is a multiple of the physical vector length. Loop-blocking refers to rewriting the loops into blocks of size `B`, to make each block accessible for later SIMDization.

In loop-level aggregation, each statement is packed with itself across consecutive iterations. Each final vector length must be a multiple of 16 bytes. A scalar `b[i]` and a vector `a[i].x,y,z` are treated alike, regardless of alignment.

Continuing with the example in short-loop aggregation (*Section 22.3.2 Phase 2: Short-Loop Aggregation* on page 646), the remaining `j` loop is SIMDized. The variable `sum` is a private variable (a variable used only in this loop), `L_result[j]` is a stride-one access, and `reduct` operates over a vector. Assuming that all the data is `int`, the blocking factor is computed as:

```
B = 16/GCD(16, sizeof(int)) = 16/4 = 4
```

where `GCD` is the Greatest Common Divisor—the largest integer that divides into both of the integer operands. SIMDization of `sum` and `L_result[j]` is straightforward. `reduct` is more complicated because its data is not stride-one across the `j`-loop. Specifically, `wt[0:39]` is now loop-invariant (its stride is 0). `dp[-j:39-j]` has a stride of -1/40. The goal is to SIMDize `reduct` into a 4-way `parallel-reduct` (see *Section 22.3.4.1 Streams* on page 648 for a definition of `parallel_reduct`, `elem_splat` and `elem_slide`).

Because `wt[0:39]` has stride 0, to transform the reduction into a 4-way parallel reduction, each of its elements needs to be replicated 4 times. A compiler-internal operation, `elem_splat(wt[0:39, 4)`, can be used for this.

For `dp[-j:39-j]`, the four vectors to be reduced in parallel are `dp[-j:39-j], ..., dp[-j-3:30-j-3]`. The corresponding elements of the four data streams form a packed vector among themselves. To produce a 4-way parallel reduction, each element of the original vector needs to be expanded by applying a sliding window of -4 elements: `elem_slide(dp[-j:39-j], -4)`. The code after loop-level SIMDization is:

```
for (j=40; j <= 120; j+=4) {
    vsum = (0, 0, 0, 0);
    vsum = vsum + parallel_reduct(+, 4, elem_splat(wt[0:39])
        *elem_slide(dp[-j:39-j], -4));
    L_result[j:j+3] = vsum;
}
```

22.3.4 Phase 4: Alignment Devirtualization

The key to automatically SIMDizing loops with arbitrary misalignment is to provide automatically generating data-reorganization operations that align data in registers. This data reorganization satisfies the alignment constraints stated above: that vector operations can only operate on aligned data, and that loads and stores to memory must be properly aligned.

As illustrated in *Figure 22-14* on page 645, the top-level algorithm for SIMDizing a loop has six phases:

- *Phases 1, 2, and 3:* SIMDize the loop as if there are no alignment constraints.
- *Phase 4:* Insert data-reorganization operations to satisfy actual alignment constraints. This phase creates the data-reorganization graph:
  - Insert the data-reorganization operations.
  - Apply optimizations to reduce the number of data-reorganization operations.
- *Phase 5:* Length devirtualization.
- *Phase 6:* Generate SIMD code. In this phase, take care of runtime alignment, unknown loop bounds, multiple misalignments, and multiple statements.

Although some alignments are known at compile-time, others are known only at runtime. In fact, runtime alignment is pervasive in applications either because it is part of the algorithms used, or because it is an artifact of the compiler's inability to extract alignment information from complex applications.

The alignment framework incorporates *length conversions*, which are conversions between data of different sizes. Length conversions are pervasive in multimedia applications, where mixed integer types are often used. Supporting length conversion greatly improves the number of SIMDizable loops.

This loop contains misaligned accesses:

```
for (i = 0; i < 100; i++) {
    a[i+2] = b[i+1] + c[i+3];
}
```

22.3.4.1 *Streams*

A *stream* is a sequence of contiguous memory locations that are accessed by a memory reference throughout the lifetime of a loop. This is also called a *memory stream*. By analogy, a stream is also a sequence of contiguous registers that are produced by an operation over the lifetime of a loop. This is also called a *register stream*.

Vector operations in a loop can be viewed as operations on streams. For example, a vector load consumes a stream of memory and produces a stream of registers. The stream is a way to view values throughout the lifetime of a loop.

An important property of a stream is its *stream offset*. This is defined as the byte-offset of the first desired value in the first register of a stream. In the presence of misaligned streams, data-reorganization operations must be inserted to enforce required stream offsets. There are several such compiler-internal operations, including:

- `vshiftstream(stream, inputOffset, outputOffset)`—Shift `stream`, the register stream, from stream offset `inputOffset` to stream offset `outputOffset`. For example, `vshiftstream(stream, 4, 0)` shifts the stream to the left 4 bytes, and `vshiftstream(stream, 0, 12)` shifts the stream to the right 12 bytes. Another way to think of this: `vshiftstream` takes an input register stream whose offset is `inputOffset`, and generates a register stream whose offset is `outputOffset`.
- `vshiftpair(v1, v2, length)`—Create a double-length vector by concatenating vectors `v1` and `v2`, and select the bytes `length, length + 1, length + 2, ... length + V - 1`, where `V` is the vector length.
- `vsplice(v1, v2, length)`—Concatenate the first length bytes of `v1` with the last (`V - length`) bytes of `v2`.
- `prependW(Stream, x)`—Prepend `x` bytes to the beginning of `Stream` of `W`-byte wide registers.
- `skipW(Stream, x)`—Skip the first `x` bytes of `Stream` of `W`-byte wide registers.
- `reduct(op, V(n, t))`—Perform operation `op` over all n elements of vector `V` and generate a single result. This is called a reduction.
- `parallel_reduct(op, x, V(n, V(x, t)))`—Perform operation `op` in parallel `x` times, over `n` distinct values. The data in the vector is interleaved: the first `x` values are the first elements in the of the `x` reductions, the next x values are the second elements in each of the reductions, and so on. The parallel reduction generates a vector of `x` reduction values.
- `vsplat(x)`—Replicate the loop invariant x across a register stream.
- `elem_splat(V(n, t), x)`—Generate a new `V(n, V(x,t))` vector where each of the values in the original vector appears `x` times, consecutively.
- `elem_slide(V(n, t), w)`—Apply a sliding window of `w` elements along the original vector.
- `vload(addr(i))`—Load a vector from a stride-one memory reference `addr(i)`. This produces a register stream whose byte-offset is the alignment of `addr(i)`.
- `vstore(addr(i), src)`—Store a vector stream from `src` to a stride-one reference at `addr(i)`.
- `vop(src1, ..., srcN)`—Take registers `src1` through `srcN` as inputs for a vector operation, and produce an output register stream. Input register streams must have matching stream offsets.
- `vpack(Stream, f)`—Pack `Stream` by the factor `f`. For example, converting from a 4-byte data type to a 2-byte data type is a stream pack with a factor of 2.
- `vunpack(Stream, f)`—Unpack `Stream` by the factor `f`. For example, converting from a 4-byte data type to an 8-byte data type is a stream unpack with a factor of 2.

In this context, *shifting left* and *shifting right* refer to data-reorganization operations that operate on a stream of consecutive registers, not to traditional arithmetic or logical shift operations.

22.3.4.2 *Zero-Shift Policy*

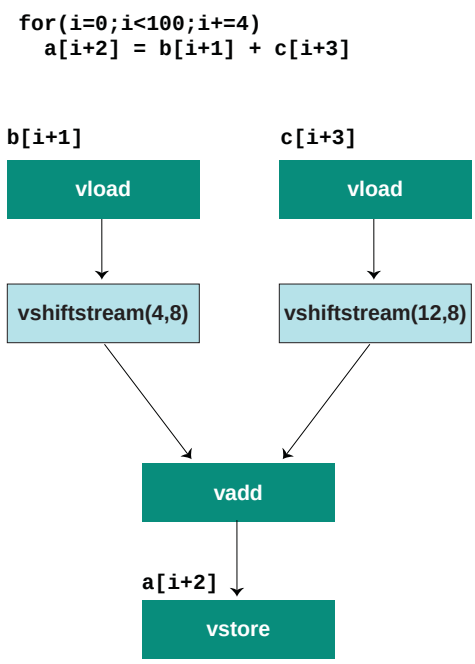
The Zero-Shift Policy is the least-optimized way to reorganize misaligned data. The idea is to shift each misaligned register stream to a stream offset of 0 immediately after it is loaded from memory. Then, each register stream is shifted to the alignment of the store address just before it is stored back to memory. Zero-Shift Policy inserts one `vshiftstream` operation for each misaligned memory stream. Zero-Shift Policy is used for runtime alignment. *Figure 22-7* on page 628 shows an example of the Zero-Shift Policy.

22.3.4.3 *Eager-Shift Policy*

The Eager-Shift Policy shifts a misaligned load stream directly to the alignment of the store, rather than to 0 as in the Zero-Shift Policy. This policy requires that alignments of loads and stores be known at compile-time. It can be more efficient than Zero-Shift Policy, lowering the number of `vshiftstream` operations needed. Eager-Shift Policy is applicable only to compile-time alignments, due to code-generation problems: the code sequence for shifting left or right is different.

Figure 22-15 shows the flow of the Eager-Shift policy, and *Figure 22-8* on page 629 illustrates the policy. It lowers the total number of stream-shift operations from 3 to 2, compared with the Zero-Shift Policy.

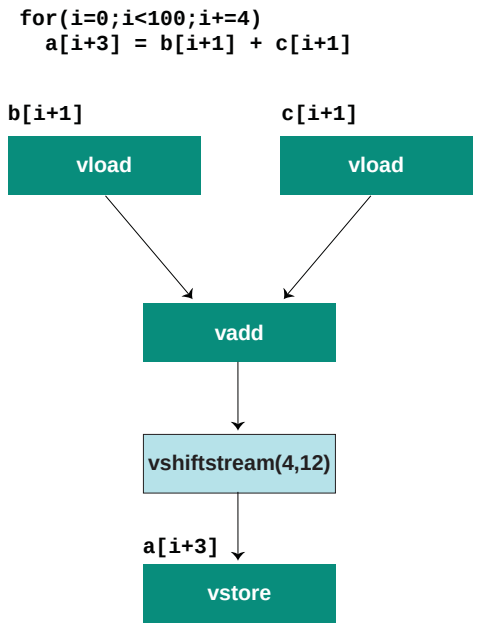
Figure 22-15. Eager-Shift Policy



22.3.4.4 *Lazy-Shift Policy*

The Lazy-Shift Policy improves upon the Eager-Shift Policy by delaying stream shifts as long as possible. For example, two relatively aligned vectors can be added by first adding them and then shifting the result, instead of first shifting them to the alignment of the store and then adding them. This helps reduce the number of `vshiftstream` operations. *Figure 22-16* on page 651 shows the lazy-shift policy. This policy exploits the case in which `b[i+1]` and `c[i+1]` are relatively aligned, and can be safely operated on as is. Only the result of the add operation needs to be shifted to match the alignment of the store.

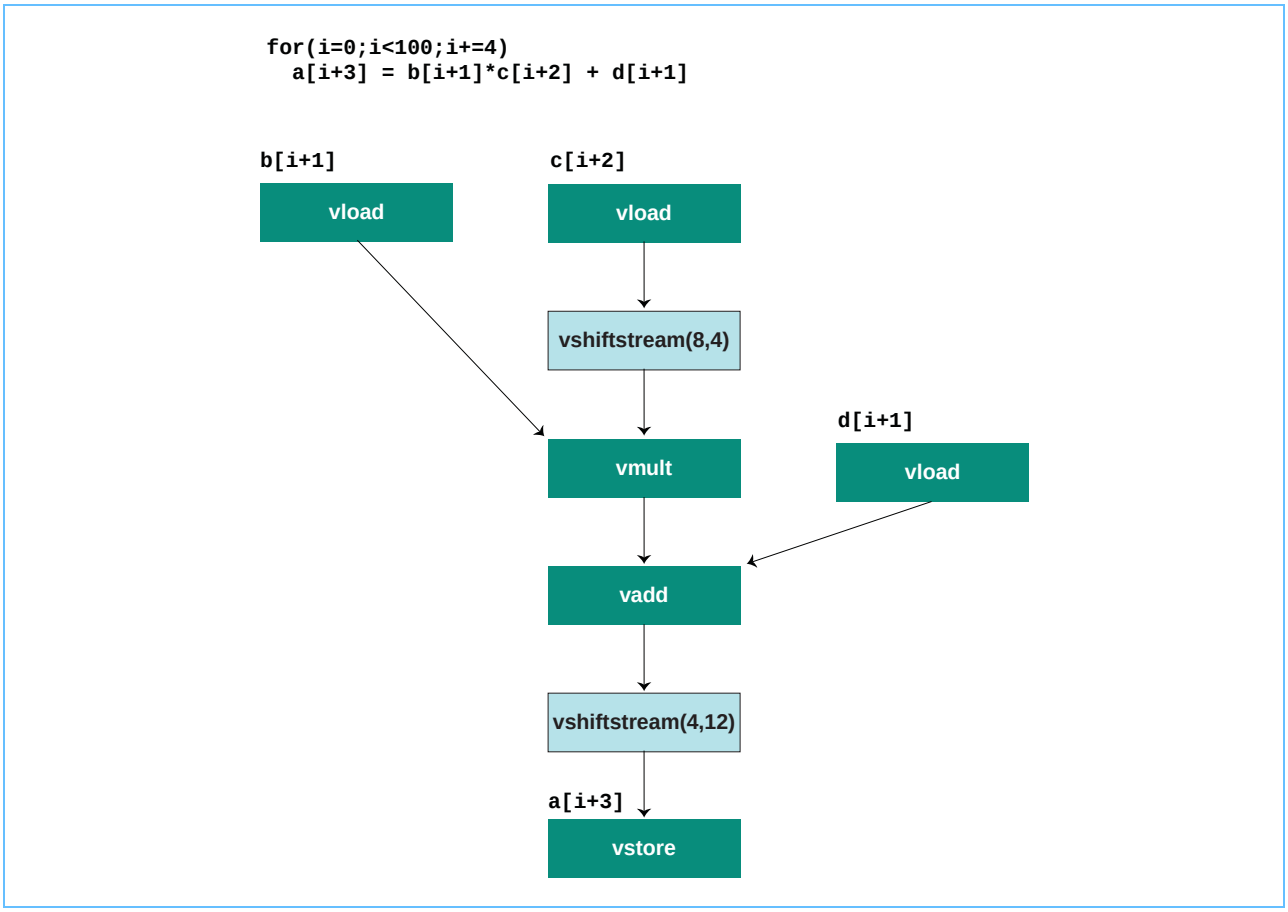
Figure 22-16. Lazy-Shift Policy



22.3.4.5 *Dominant-Shift Policy*

The Dominant-Shift Policy reduces the number of `vshiftstream` operations the most. It is most effective if applied after the Lazy-Shift Policy. The offset that is dominant (most often used) among a set of streams is chosen as the offset to shift to. For example, if 4 is the offset used most often, then 4 is the dominant offset, and chosen as the shift target. In *Figure 22-17* on page 652, the dominant offset is the stream offset 4. Shifting the stream to this offset decreases the number of `vshiftstreams` from 4 (for the Zero-Shift Policy) to 2.

Figure 22-17. Dominant-Shift Policy



22.3.4.6 *Length Conversion*

A length conversion results from any operation that converts a value of one length to a value of another length. The most common length conversions occur between data types of different sizes. Such data conversions are common in multimedia workloads because multimedia data, such as pixels or colors, are stored in compact forms like `short` or `char`, but are operated on as integers or floats to reduce rounding errors. The data-reorganization operations, `pack` and `unpack`, are used to handle data of different lengths. All four shift policies can be applied to expression trees with length conversion. The data-reorganization graph created by Phase 4 (*Section 22.3.4* on page 648) is a usual compiler expression tree augmented with data-reorganization operations.

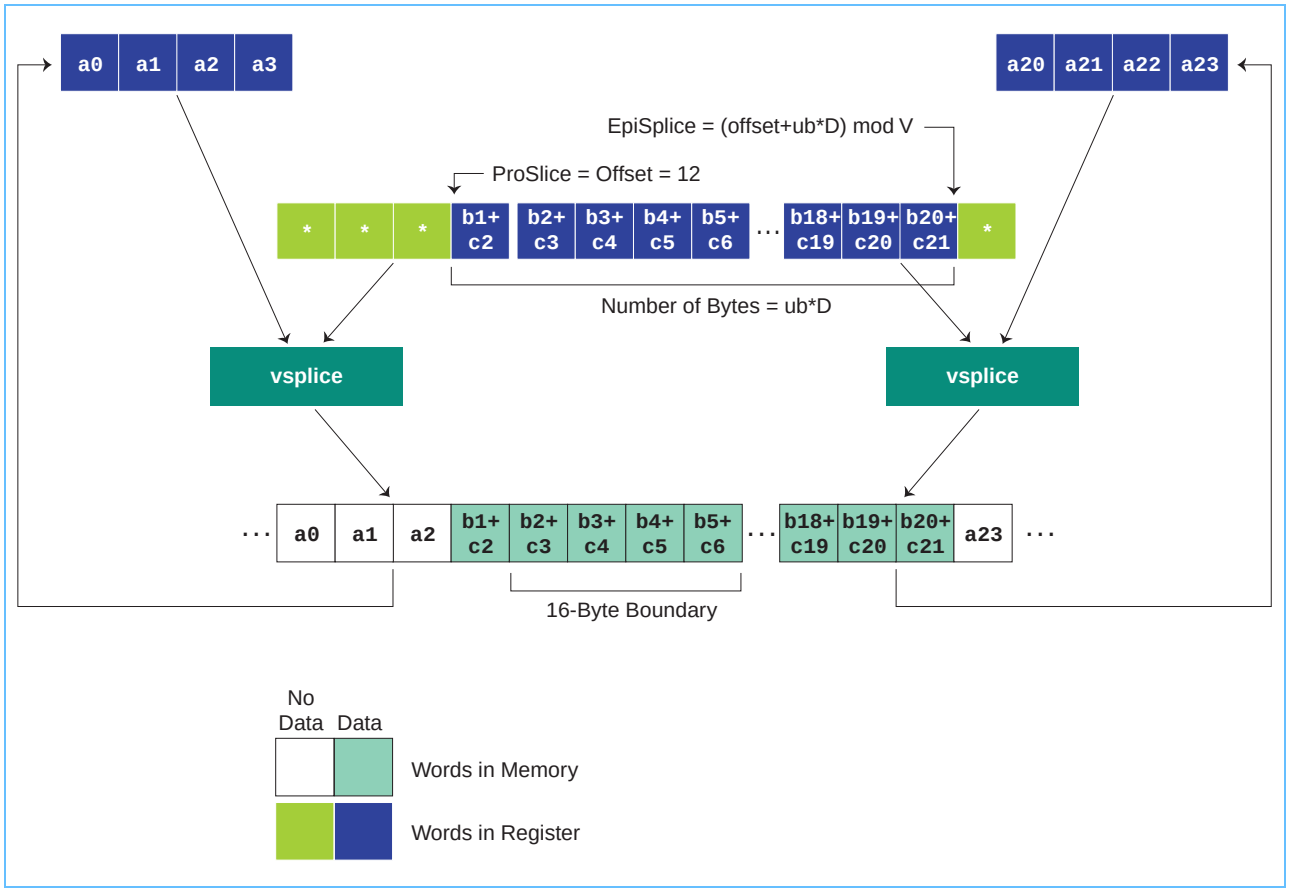
22.3.4.7 *Handling a Partial Store*

The first and last iterations of a loop often need to be handled separately, in a prolog (*Section 22.1.2.6* on page 632) and epilog (*Section 22.1.2.7* on page 633).

Partial stores are implemented by loading the original value prior to the store, splicing it with the newly computed value, and then storing the spliced value back into memory, using the `vsplice` operation. *Figure 22-18* shows how partial stores are implemented by first loading the original value before the store. Then, this value is spliced with the newly computed value. Finally, the spliced value is stored back into memory, using the compiler-internal function, `vsplice`. At the end of Phase 4, the vectors are aligned, and their lengths are multiples of the physical vector length.

The Vector/SIMD Multimedia Extension store vector left and right instructions (**lvlx[l]**, **lvrx[l]**, **stvlx[l]**, and **stvrx[l]**) can also be useful for simplifying partial-vector stores.

Figure 22-18. Implementing Partial Vector Store



22.3.5 Phase 5: Length Devirtualization

In Length Devirtualization, virtual vectors are first flattened to vectors of primitive types. Then, operations on the virtual vectors are either mapped to operations on multiple physical vectors, or rewritten as operations on scalar values. The decision of whether to map a virtual vector onto physical vectors or onto scalars is based on the length of the virtual vector and other heuristics.

At the end of Phase 5, all vectors have become physical vectors. They are aligned and operate on 16-byte data.

22.3.6 Phase 6: SIMD Code Generation and Instruction Scheduling

22.3.6.1 *Code Generation*

The Code Generation phase maps the SIMDized operations, including the data-reorganization operations, to SIMD instructions specific to the target platform (either Vector/SIMD Multimedia Extension or SPE). Code generation takes the augmented expression tree (the expression tree plus data-reorganization operations) as input, and maps the generic stream-shift operations to native SIMD permutation instructions. *Figure 22-19* on page 655 shows the basic mechanism to shift a stream with offset 4 to offset 0. In this case, the stream is shifted left by 4 bytes. The compiler-internal function, `vshiftstream`, is mapped to a sequence of shift operations across pairs of vector registers. These shift operations are called `vshiftpair` operations; `vshiftpair` is another compiler-internal function.

The code-generation and earlier phases of the framework also tackle four difficult problems:

- Runtime alignment.
- Unknown loop bounds.
- Multiple misalignments.
- Multiple statements (a loop body that contains more than one statement).

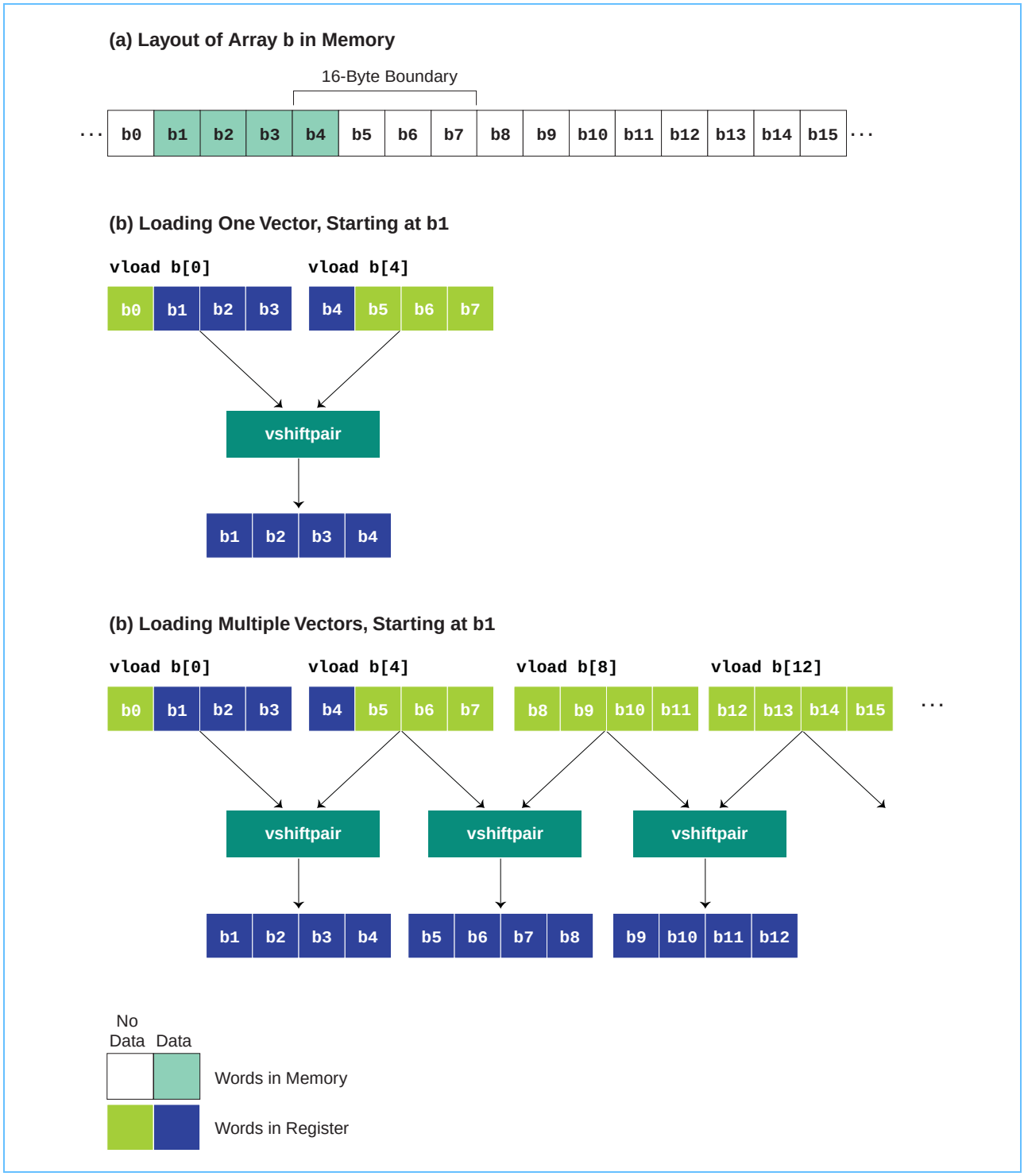
When memory alignment cannot be known at compile-time, it is referred to as *runtime alignment*. Runtime alignment can be inherent to a program, or it can result when the program uses complex pointer arithmetic. To handle runtime alignment, transform the original memory streams and computation streams into equivalent streams that can be shifted to arbitrary alignments. An arbitrary stream-shift, denoted by `vshiftstream(stream, left, right)`, can be converted to a left stream-shift to the required offset. The derived stream starts exactly `left` bytes before the first value of `stream`.

Here are some guidelines for applying data-reorganization operations for runtime alignment:

- Lazy-Shift, Eager-Shift, and other policies can be applied to runtime alignment.
- When propagating stream offsets, that is, propagating the alignment of the store to the load during an Eager-Shift, the stream offset needs to be scaled up or down. The stream offset is scaled by the packing factor.
- Avoid placing a stream-shift that causes an unpack between a register stream and a memory stream, with no other stream-shifts in between.
- When possible, place stream-shifts at the shorter end of a data conversion. It takes fewer machine instructions to shift a stream of shorter length.

22.3.7 SIMDization Example: Multiple Sources of SIMD Parallelism

Figure 22-19. Implementing Stream Shift



This section demonstrates how to extract parallelism at the basic-block and loop levels. The original code sample, shown below, is a simple loop with four statements. This code is followed by

three other samples—basic-block aggregation, loop aggregation, and length devirtualization, also shown below. These code samples are truncated, so that they highlight the data elements on which we focus.

We focus only on the stores generated by each of the loop's statements, which are illustrated in *Figure 22-20* on page 657, although the compiler would take all memory references into account. Each 4-byte store is represented by a box in *Figure 22-20*. The sets of boxes alternative in color to distinguish consecutive iterations of the original loop.

Original Code:

```
for (i = 0; i < n; i++) {
    a[i].x =
    a[i].y =
    a[i].z =
    b[i] =
}
```

Step 1: Basic-Block Aggregation:

```
for (i = 0; i < n; i++) {
    (a[i].x,y,z)   =
    b[i]           =
}
```

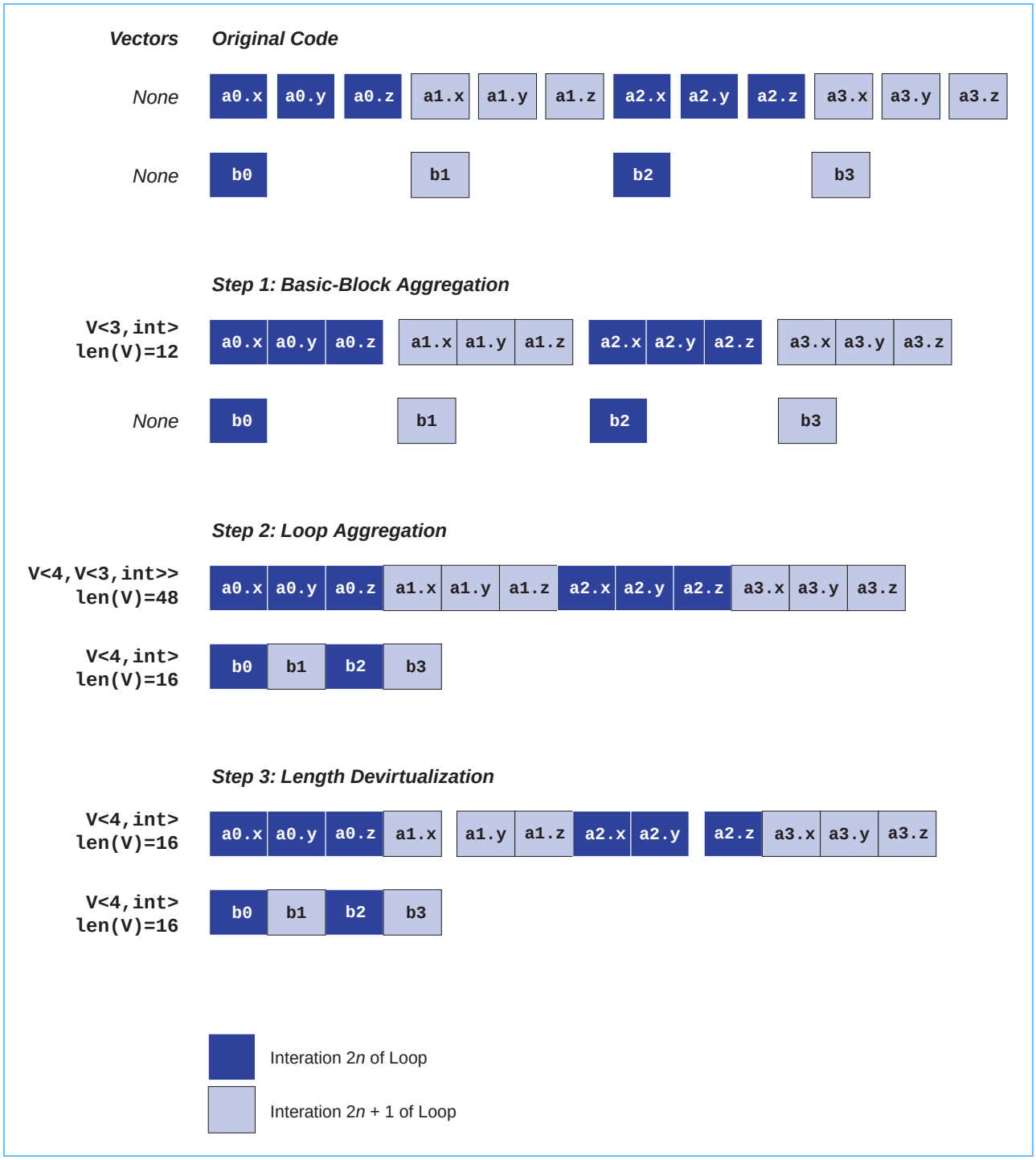
Step 2: Loop Aggregation:

```
for (i = 0; i < n; i+=4) {
    (a[i].x...a[i+3].z)   =
    (b[i].......b[i+3])   =
}
```

Step 3: Length Devirtualization:

```
for (i = 0; i < n; i+=4) {
    (a[i].x.....a[i+1].x) =
    (a[i+1].y...a[i+2].y) =
    (a[i+2].z...a[i+3].z) =
    (b[i].......b[i+3])   =
}
```

Figure 22-20. SIMDization Example with Mixed Stride-One and Adjacent Accesses



The first step is basic-block aggregation. The compiler recognizes that the first three stores (`a[i].x a[i].y, a[i].z`) are adjacent in memory. Assuming the right-hand sides of the three statements are isomorphic, the compiler aggregates the three statements into a vector of three integers stored into `a[i].x, y, z`. The `b[i]` statement remains unchanged.

The second step is loop-level aggregation. Recognizing the vector store `a[i].x,y,z` and the element store `b[i]` as stride-one accesses, the compiler aggregates these accesses across loop iterations. It treats the new vector `a[i].x,y,z` statement no differently than any other statements in the loop. The only difference between the `a[i].x,y,z` and `b[i]` statements is that the first one generates a 12-byte value but the second one generates a 4-byte value.

During loop-level aggregation, the compiler extracts SIMD parallelism from the smallest number of consecutive iterations, while ensuring that each vector in the loop has a length that is a multiple of the physical vector length: 16 bytes. The optimal blocking factor is 4, since it aggregates 4 of the 12-byte `a{i}.x,y,z` vectors into a new compound vector of 48 bytes. Four of the 4-byte `b[i]` values aggregate into a new vector of 16 bytes.

An alignment-devirtualization step may be needed. This step is necessary when arrays `a` and `b` are not aligned. Since both `a[i].x ... a[i+3].z` and `b[i] ... b[i+3]` are stride-one accesses, their alignment constraints are handled no differently than any stride-one access of native data types. For simplicity, our example and *Figure 22-20* assumes both vectors are aligned. But suppose that one of them were not; for example. suppose a[i].y is 16-byte aligned instead of `a[i].x`. Then, the alignment devirtualization phase would logically skew the computation associated with vector `a` to the right by one value. As a result, the blocked loop would effectively compute and store the `a[i].y ... a[i+4].x` instead of `a[i].x ... a[i+3].z`.

The third step (in this example) is length devirtualization. The long `a[i].x ... a[i+3].z` vector is broken down to three physical vectors. Each of the three physical vectors mixes original stores from two consecutive iterations. At this stage, all vectors have become physical vectors, meaning that they are aligned and operate on 16-bytes of data.

In this example, the basic-block and loop-level aggregation steps each handle some aspect of SIMDizing the loop, but neither could SIMDize the loop alone:

- On one hand, basic-block SIMDization combines the otherwise non-stride-one accesses `a[i].x, a[i].y, a[i].z` into stride-one accesses that can be subsequently exploited in loop-level aggregation.
- On the other hand, loop-level SIMDization aggregates the unaligned vector of 12-bytes, `a[i].x,y,z` produced by basic-block aggregation into an aligned vector of a multiple of the physical vector length.

This is achieved transparently within the proposed SIMDization framework without expensive loop restructuring such as loop distribution, loop rerolling and collapsing, and loop unrolling.

22.3.8 SIMDization Example: Multiple Data Lengths

There are two types of loops that mix data of different lengths—for example, `char`, `short int`, and `float`. The first type has homogenous statements: statements that each operate on data of uniform length. Some of the homogenous statements do operate on types of different lengths. The second type has statements that operate on distinct data lengths within a unique statement. This code example demonstrates such heterogeneous statements:

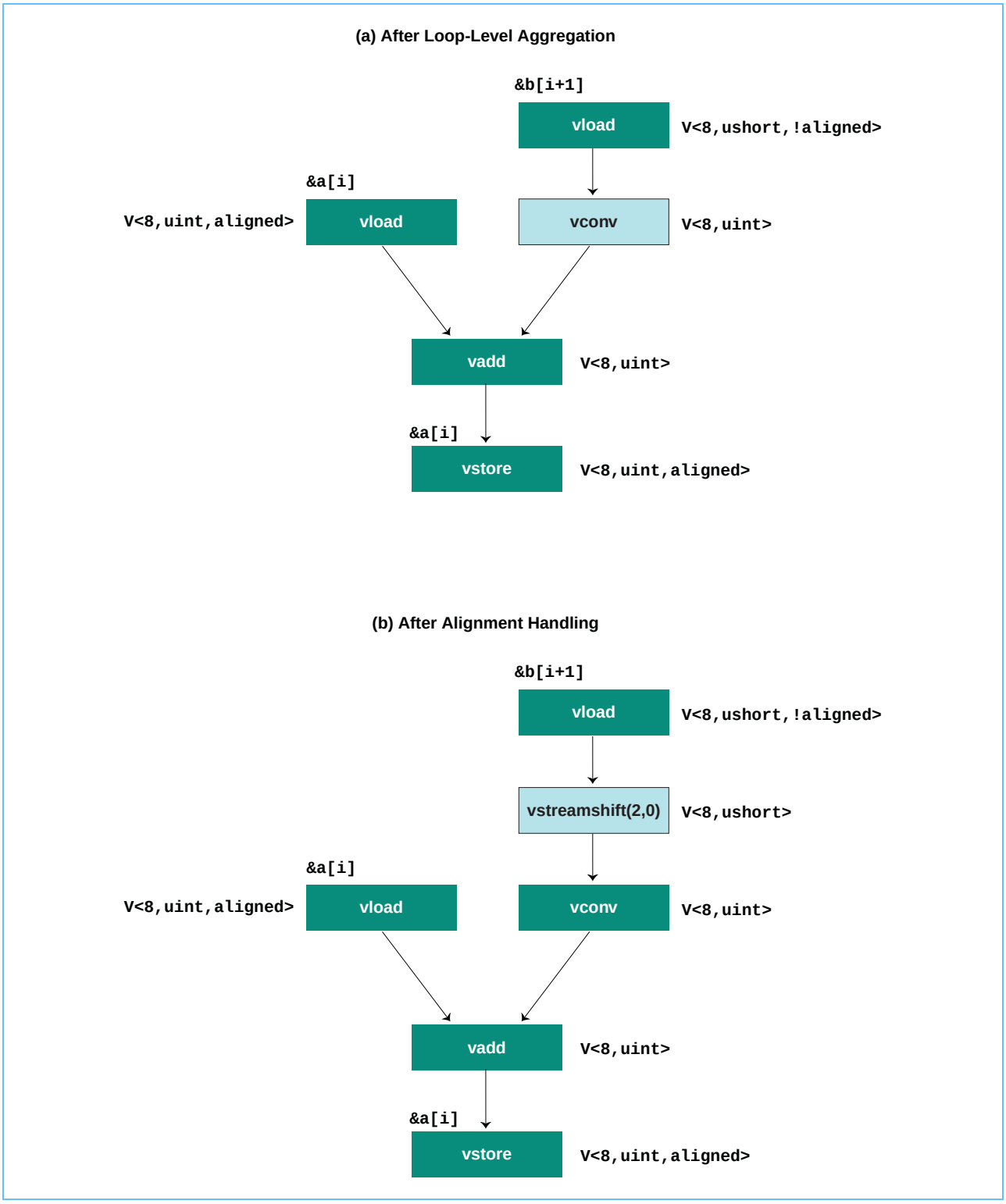
```
uint a[N];
ushort b[N];
for (int i = 0; i < N; i++) {
    a[i] += (uint) b[i+1];
}
```

Such heterogeneous statements often require conversion operations between data of different lengths. This is referred to as *length conversion*. Loops that require length conversion among heterogeneous statements are more constraining than loops that contain homogenous statements. The complexity is due to the fixed-length of SIMD vectors. For example, a SIMDized length conversion from 1-byte length to 2-byte length consumes one vector and produces two vectors. Consuming one vector but producing multiple vectors (or a fraction of a vector, if the conversion is the other way around) is unnatural for most compiler intermediate representations. The example that follows focuses on the more challenging length conversion among heterogeneous statements.

22.3.8.1 ***Alignment Handling and Alignment Devirtualization***

Figure 22-21 on page 660 shows the expression tree for the above code after the first three phases (basic-block, short-loop, and loop-level aggregation).

Figure 22-21. Length Conversion After Loop-Level SIMDization



In the tree representation, each node is labeled with the tuple `V(n, t, aligned)` to represent the number of elements, the type, and the alignment properties of each vector. Since `aligned` only affects vector load and store semantics, it is specified only for memory nodes.

For alignment devirtualization, each array accessed in the loop must be stride-one in order to form a stream of contiguous memory accesses across iterations. The alignment-devirtualization algorithm is mainly concerned with code generation problems using stream-shift operations with length conversion.

In our example, assume the base addresses `a` and `b` are aligned. Assume the streams represented by `a[i]` and `b[i+1]` have an offset of 0 and 2, respectively. These streams are therefore relatively misaligned. *Figure 22-21(b)* shows the alignment tree after alignment handling. The node `vstreamshift(2, 0)` represents a pseudo operation (a compiler-internal function) that shifts the entire input stream of `b[i+1]` with an offset of 2 to 0, to match the offset of the `a[i]` stream.

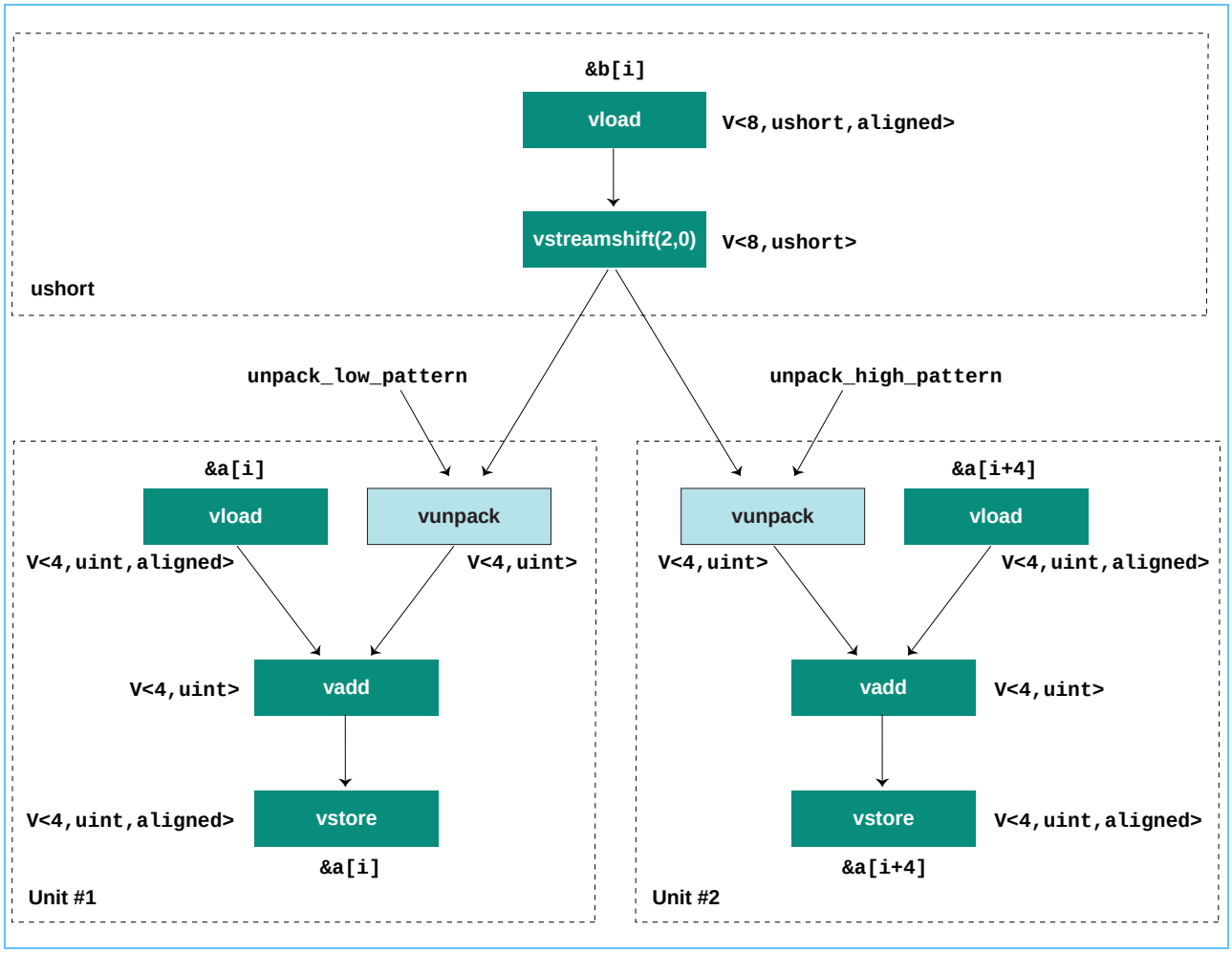
22.3.8.2 *Length Devirtualization*

For most vector operations, devirtualization simply involves breaking a vector into multiple physical vectors. Length conversion, however, requires special handling because a devirtualized length conversion may consume one physical vector and produce multiple physical vectors, or vice versa. The two generic operations, `pack` and `unpack`, are used to express such data reorganization:

- `pack(v1, v2, vpattern)`—This operation produces a physical vector by selecting bytes from physical vector `v1` and `v2` according to the byte pattern specified by physical vector `vpattern`.
- `unpack(v, vpattern)`—This operation produces a physical vector by selecting bytes from physical vector `v` according to the byte pattern specified by physical vector `vpattern`.

In our example, the conversion operation unpacks a vector of `short` (16-bit integers) into two vectors of `int` (32-bit integers). *Figure 22-22* on page 662 shows the expression trees after devirtualization. This SIMDized code consists of three distinct parts. The top part represents the computations done with the variables of type `short`, namely loading from memory. The bottom two parts represent the computations done with the variables of type `int`, with the left and right bottom parts each consuming half the data produced by the top part.

Figure 22-22. Length Conversion After Devirtualization



22.3.8.3 *SIMD Code Generation*

The generic `pack` and `unpack` operations are mapped to instructions of the target platform. Integrating conversion-handling into the SIMDization framework is made much easier because stride-one accesses are preserved in early phases of SIMDization. As shown in *Figure 22-22*, when devirtualizing the `short` to `int` conversion, the stride-one access `a[i]` in the original loop becomes two non-stride-one vector accesses, `a[i:i+3]` and `a[i+4:i+7]`, in the SIMDized loop. Since stride-one accesses are critical both to aggregation phases and alignment handling, by maintaining the stride-one accesses until devirtualization, the early SIMDization phases can process length-conversion in the same way that other regular operations are processed.

22.3.9 Vector Operations and Mixed-Mode SIMDization

The concept of virtual vectors and devirtualization—Phase 4 in *Section 22.3.4* on page 648, and Phase 5 in *Section 22.3.5* on page 653—can be extended to vector operations. During early phases of SIMDization, a vector operation can be SIMDized even if there is no direct support in the underlying SIMD hardware. When these virtual vector operations are devirtualized, they can be replaced by:

- A single SIMD instruction.
- A sequence of SIMD instructions.
- A library call.
- A sequence of scalar operations.

This process is called *mixed-mode SIMDization*. A loop that contains computation that can be executed on the scalar units, or on the SIMD unit, or both, is a candidate for mixed-mode SIMDization. For example, on the PPE's Vector/SIMD Multimedia Extension unit, double-precision floating-point instructions are executed in scalar units, but saturate-arithmetic is executed on the SIMD (VXU) unit. Most fixed-point instructions can execute on both units. Since the VXU unit is more constrained than the scalar unit, it can be more efficient to compute in scalar mode if accesses are misaligned or not stride-one.

Loop distribution is one way in which heterogeneous loops (loops containing a mixture of scalar and vector computation) can be handled. Loop distribution may cause large loops be distributed into smaller ones. More loops with shorter loop bodies works against finding instruction-level parallelism, because it decreases the size of basic blocks. Overly aggressive loop distribution is one of the major causes of SIMDization-related performance degradation.

Mixed-mode SIMDization avoids the problem of loop distribution affecting ILP, and may help ILP as the scheduler can use the SIMD and scalar units (and registers) at the same time. It is not always straightforward to decide which mix of SIMD and scalar computation is best. Both hardware constraints and resource constraints of the currently executing program need to be taken into account. In addition, the VXU and scalar units can be used for the same operation at the same time.

Virtualizing vector operations also provides a way to change transformations made earlier, if during later analysis, when more information is available, it becomes clear that those transformations did not improve the code. For example, during basic-block SIMDization, non-stride-one memory accesses may be packed into vectors, hoping that these vectors become stride-one after loop-level aggregation. But if that does not happen and the packing overhead is high, the compiler can undo the overly optimistic packing decision by devirtualizing the vector operation back to scalar operations.

22.4 Other Compiler Optimizations

22.4.1 OpenMP

A compiler can support OpenMP directives and routines. OpenMP is a platform- and language-neutral way for programmers to insert parallelization directives in source code. OpenMP is defined by a consortium of universities and vendors, and the specification is found at http://www.openmp.org. See *Section 21 Parallel Programming* on page 599 for a discussion of how to use OpenMP constructs in programs run on the CBE processor.

Most of the OpenMP directives are orthogonal to SIMDization, and they should not interfere with SIMDization. Some OpenMP directives do parallelize loops in the way that SIMDization does, and there can be some side effects. If the OpenMP directives are processed first, the loop structures can be perturbed in a way that prevents SIMDization. One of the OpenMP directives is the *parallel for* construct. This is used to parallelize a simple loop. For example:

```
void demo(int n, float *a, float *b) {
    int i;

#pragma omp parallel for

    for (i = 1; i < n; i++)  /* i is private, by definition */
        b[i] = (a[i] + a[i-1]) / 2.0;
}
```

It is most important for SIMDization that OpenMP not corrupt the alignment information. When compile-time alignment is available for arrays before OpenMP directives are processed, this compile-time alignment should remain in force after processing OpenMP directives. Whenever possible, keep array subsections in multiples of 16 bytes.

22.4.2 Subword Data Types

Both 2-byte and 1-byte data types are called *subword* data types. The C programming language data types `char` and `short` are 1 and 2 bytes, respectively. From the architecture's perspective, not all data types are equally supported. For example, the SPU ISA provides 16-bit multiply only, so 32-bit multiply is implemented in software using 16-bit multiplies. Subword data types are commonly used in multimedia and games programs, so the SIMD units are designed to provide better support for subword data types.

The C programming language requires that subword data types be promoted to a full word before being operated on. A word is 32-bits or 4-bytes; the C data type `int` corresponds to a word. C allows a programmer to declare a subword variable, in terms of alignment, size and layout in memory—but there is no subword arithmetic. When a `short` or `char` is loaded into a register, the values are converted into 32-bit integers by sign-extensions. Without additional optimizations, there is no way to generate subword instructions directly from scalar C source programs.

Subword arithmetic optimizations bridge the gap between C's inability to specify subword arithmetic and the underlying hardware's strength in supporting subword operations. There are two steps:

- *Eliminate Redundant Partial-Copy Operations (from sign extensions).* This relies on partial-copy propagation. Because subword instructions use only a partial region of a register and sign-extension instructions copy only a partial region of a register, redundant sign-extensions are readily removed. In addition, this step performs general constant folding. Constant folding refers to the compiler precalculating constant expressions.
- *Generate Subword Arithmetic Operations.* This converts integer arithmetic operations to the equivalent arithmetic operation on narrower-width data types. This produces more efficient code when the ISA provides subword instructions, but not the equivalent instructions on 32-bit data types. In addition, since this optimization narrows the operating data, it exposes more opportunity for eliminating redundant sign extensions.

22.4.3 Backend Scheduling for SPEs

For SPEs, there are additional code-generation issues that must be handled for both automatically SIMDized instructions and scalar instructions. Instruction scheduling occurs just before the compiler emits the final assembly or object code. The scheduling process consists of two closely interacting subtasks: scheduling and bundling. While typical schedulers deal with resources and latencies, a highly optimizing scheduler takes into account constraints that are expressed in numbers of instructions. For example, the HBR branch hint instruction cannot be more than 256 instructions away from its target branch, and should be no closer than 8 instructions. (See *Section 24.3.3 Branch Hints* on page 691.) The scheduling is complicated by the fact that the number of instructions in a scheduling unit is known only after the bundling subtask has completed.

The bundler's main role is to ensure that each pair of instructions that are expected to be dual-issued satisfies the SPE's instruction-issue constraints. The processor will dual-issue independent instructions only when the first instruction uses the even pipeline and resides in an even word address, and the second instruction uses the odd pipeline and resides in an odd word address. After instruction ordering is set by the scheduling subtask, the bundler can only impact the even or odd word address of instructions by inserting `nop` instructions into the instruction stream.

Another important task of the bundler is to prevent instruction-fetch starvation. Recall that a single local memory port is shared by the instruction fetch mechanism and the processor's memory instructions. As a result, a large number of consecutive memory instructions can stall instruction fetching. With 2.5 instruction-fetch buffers reserved for the fall-through path, the SPE can run out of instructions in as few as 40 dual-issued cycles. When a buffer is empty, there may be as few as 9 cycles for issuing an instruction-fetch request to still hide its full 15-cycle latency. Since the refill window is so small, the bundling process must keep track of the status of each buffer and insert explicit `ifetch` instructions when a starvation situation is detected.

The refill window is even smaller after a correctly-hinted branch that is taken, since there is only one valid buffer after a branch. This compares to 2.5 buffers for the fall-through path. Instruction starvation is prevented only when all instructions in the buffer are useful. That is, the branch target must point to an address that is a multiple of 16 instructions, which is the alignment constraint of the instruction-fetch unit. This constraint is enforced by introducing additional `nop` instructions before a branch target, to push it to the next multiple of 16 instructions. These compiler heuristics are successful in scavenging any idle issue slots, so that `nop` instructions may be inserted without performance penalties.

A final concern of the bundling process is to make sure that there is a sufficient number of instructions between a branch hint and its branch instruction. This constraint is due to the fact that a hint is only successful if its target branch address is computed before that branch enters the instruction-decode pipeline. The bundler adds extra `nop` instructions when the schedule did not succeed in interleaving a sufficient number of independent instructions between a hint and its branch.

A unified scheduling and bundling phase produces best performance. The compiler can preserve the cycle scheduling approach, in which each non-decreasing cycle of the schedule is filled in turn. The exception is that the bundling process may retroactively insert `nop` or `ifetch` instructions. When scheduling the next cycle in the scheduling unit, the compiler checks to determine whether an `ifetch` instruction is required. If it is, an `ifetch` instruction is inserted, and the scheduling resource model is updated. The normal scheduling process then continues. When no additional instructions can be placed in the current cycle, the compiler checks to determine whether `nop` instructions must be inserted in previous cycles to enable dual-issuing.

22.4.4 Interacting with Typical Optimizations

SIMDization should occur after aggressive dependence analysis and high-level loop transformations. These loop transformations enhance SIMDization by removing loop-carried dependences among inner-most loops:

- Loop interchange, which swaps an inner (nested) loop with an outer loop. This is done when the inner loop appears to be the candidate for SIMDization.
- Loop distribution, which splits off those parts that cannot be SIMDized.
- Loop versioning, a technique for hoisting an individual array index exception check outside a loop by providing two copies of the loop: the safe loop and the unsafe (original) loop.

SIMDization can be followed by other loop transformations, including:

- Loop fusion.
- Loop unrolling, which removes needless copy operations.
- Predictive commoning, a special form of common subexpression elimination that exploits data reuse among consecutive loop iterations.

Some optimizations are not suitable for SIMDization:

- Loop rerolling converts basic-block parallelism to loop-level parallelism by rerolling statements into loops. The drawback is that a new inner-most loop is added to the loop nest. Since loop-level SIMDization happens at the inner-most loop, rerolling does not expand the scope of SIMDization. In addition, rerolling requires a loop counter, which is not always known.
- Loop collapsing rerolls statements into a loop and then collapses it to the inner-most surrounding loop. Loop collapsing exploits parallelism at both the basic-block level and the loop level. However, it is more constraining since it requires the original inner-most loop to be perfectly nested in its surrounding loop, and it requires all its statements to be rerolled.
- Loop unroll-and-pack converts loop-level parallelism to basic-block parallelism by unrolling the loop and packing the computation in a basic block with SIMD instructions. Since SIMDization is limited to a basic block in this case, it is very inefficient to handle misalignment

and reduction. In addition, packing algorithms are more expensive and randomized, and sometimes perform less well, than loop-level SIMDization based on dependence vectors.

23. Vector/SIMD Multimedia Extension and SPU Programming

Both the PPE and the SPEs support very similar sets of vector instructions. The PPE processes SIMD operations in the Vector/SIMD Multimedia Extension unit within its PowerPC Processor Unit (PPU). The operations are those of the Vector/SIMD Multimedia Extensions to the PowerPC instruction set. The SPEs process SIMD operations in their Synergistic Processing Unit (SPU). The operations are those of the SPU instruction set. Both instruction sets operate on 128-bit SIMD vectors and both support many of the same types of SIMD operations.

This chapter summarizes key similarities between these instructions and intrinsics, with the goal of identifying strategies for developing code that is portable between the PPE and the SPEs.

23.1 Architectural Differences

The objectives of the Vector/SIMD Multimedia Extension and SPU architectures are essentially the same—to perform well on data-intensive processing, such as graphics pipelines, stream processing (encoding, decoding, encryption, decryption), and physical modeling. The architectural resources, however, that each architecture brings to bear differ in some important respects. Some of the major SIMD-support differences between the PPE and SPE architectures are summarized in *Table 23-1*.

Table 23-1. PPE and SPE SIMD-Support Comparison

Feature	PPE	SPE
Number of processing elements	1	8
Modes supported	user and supervisor	user only
Number of SIMD registers	32 (128-bit)	128 (128-bit)
Organization of register files	separate fixed-point, floating-point, and SIMD registers	unified SIMD registers
Load latency	variable (cached)	fixed
Addressability	2^{64}-byte main storage	256-KB LS 2^{64}-byte main storage via DMA
Memory architecture	2-level caching	software-controlled LS
SIMD instruction set	general SIMD, supported by PowerPC scalar and control instructions	SIMD only, optimized for single-precision floating-point, 16-bit fixed-point, and 32-bit fixed-point
Single-precision floating-point SIMD	IEEE 754-1985 and SPE-compatible graphics-rounding mode supported	extended range[1]
Double-precision floating-point SIMD	not supported	IEEE 754-1985 supported
Doubleword fixed-point SIMD	not supported	supported
Inter-element communication facility	MMIO	channel

1. See *Section 3.1.4* on page 68.

The PPE excels at control processing. Unlike the SPEs, which supports only problem (user) state, the PPE supports problem (user) state, privileged (supervisor) state, and hypervisor state. All of the PPE instructions—standard PowerPC and Vector/SIMD Multimedia Extensions—can

be intermixed in a single program.The PowerPC and Vector/SIMD Multimedia Extensions architectures are also supported by well-developed tools for debug and performance analysis, although this advantage is changing as new tools become available for the SPE. Perhaps most importantly, there is a substantial quantity of PowerPC and Vector/SIMD Multimedia Extension legacy code.

The eight SPEs provide a significant boost in execution resources for both scalar and SIMD processing, as compared to the single PPE. SPU instructions provide excellent support for single-precision floating-point graphics applications. Several SPU instructions are available for eliminating inefficient execution penalties usually associated with branch execution. Blocking channels for communication with the MFC or other parts of the system external to the SPU, provide an efficient mechanism for waiting on the completion of external events without the need for polling or interrupts, both of which burn power needlessly.

23.1.1 Registers

23.1.1.1 *PPE Registers Used by Vector/SIMD Multimedia Extension Instructions*

The PPE has 32 Vector Registers (VRs) for holding Vector/SIMD Multimedia Extension instruction operands, as described in *Section 2.3* on page 48. It also has 32 General-Purpose Registers (GPRs) and 32 Floating-Point Registers (FPRs) for holding fixed-point scalar operands and floating-point scalar operands, respectively.

In addition to these instruction-operand register files, the PPE also has three control and status registers that are affected by Vector/SIMD Multimedia Extension operations—a 32-bit Vector Status and Control Register (VSCR), a 32-bit Vector Save Register (VRSAVE), and a 32-bit (8-element) Condition Register (CR). The VSCR register has two defined bits—non-Java-mode and saturation [SAT]. The VRSAVE register may be used in accordance with the ABI for saving and restoring Vector/SIMD Multimedia Extension state across context switches. The CR6 field of the CR register is useful for branch control; the bit is set or cleared according to the result of a Vector/SIMD Multimedia Extension compare instruction that is of the recording form (a compare instruction that has a dot suffix on the mnemonic).

The VSCR[SAT] bit is set if saturation (an inexact result) occurs during a saturating SIMD operation. To use this bit, the contents of the VSCR can be copied to a VR (using the **mfvscr** instruction), bits other than the SAT bit can be cleared in the VR (**vand** with a constant), the result can be compared to zero so as to set CR6 (**vcmpequb.**), and a PowerPC conditional branch instruction can test CR6.

The PPE's Floating-Point Status and Control Register (FPSCR) is not affected by Vector/SIMD Multimedia Extension floating-point operations. In general, no status bits are set to reflect the results of such operations, except that VSCR[SAT] bit may be set by the Vector Convert To Fixed-Point Word instructions.

23.1.1.2 *SPE Registers*

Each SPE has 128 registers in one unified register file that holds all types of operands and instructions. In addition, the SPE also has a 128-bit Floating-Point Status and Control Register (FPSCR) that is updated after every floating-point operation to record information about the result and any associated exceptions.

The SPE architecture does not have a condition register comparable to the PPE's CR register. Instead, SPE comparison operations set result fields in the result vector that are either 0 (false) or -1 (true) and that are the same width as the operand elements being compared. These results can be used for bitwise masking, select instructions, or conditional branches.

The SPU instruction set includes many conditional branch instructions. An SPE conditional branch requires moving the vector element containing the branch condition into the preferred slot using a quadword rotate, performing a compare, and branching based on the result of the compare. By comparison, a Vector/SIMD Multimedia Extension conditional branch requires a compare, a mask, a compare of CR6, and then a conditional branch, which is more expensive than an SPE's conditional branch.

23.1.2 Data Types

For the most part, Vector/SIMD Multimedia Extension instructions operate on 128-bit vectors whose elements are byte, halfword, or word data types. The only area in which they do not are in the sourcing of a pointer from a GPR and writing a predicate[54] result to a GPR.

To this list of data types, the SPU architecture adds the doubleword data type. *Table 23-2* summarizes the supported vector data types for each architecture.

Table 23-2. Vector/SIMD Multimedia Extension and SPU Vector Data Types

Data Type	Vector/SIMD Multimedia Extension	SPU
vector char (signed and unsigned)	yes	yes
vector bool char	yes	no[1]
vector short (signed and unsigned)	yes	yes
vector bool short	yes	no[1]
vector pixel	yes	no
vector int (signed and unsigned)	yes	yes
vector bool int	yes	no[1]
vector float	yes	yes
vector long long (signed and unsigned)	no	yes
vector double	no	yes

1. The *SPU C/C++ Language Extensions* specification does not define specific boolean vector types. Boolean vectors are supported as unsigned vectors.

Among the important differences are:

- Only the Vector/SIMD Multimedia Extension instructions supports pixel vectors.
- Only the SPU instructions support doubleword vectors.

54. A predicate is Boolean-logic term denoting a logical expression that determines the state of some variables. All the Vector/SIMD Multimedia Extension predicates return their results to a GPR. They do this by performing a vector compare and then moving and masking CR6 to a GPR.

The SPUs quadword data type is excluded from the list because it is a non-specific vector data type instead of a specific vector data type. The quadword data type is used exclusively as an operand in specific intrinsics—those which have a one-to-one mapping with a single assembly-language instruction.

23.1.3 Instruction-Set Differences

The Vector/SIMD Multimedia Extension and SPU instructions were architected to integrate seamlessly, with an emphasis on frequent and important data types. Thus, they have many similarities, in addition to their use of 128-bit vector operands. Both architectures support a large collection of fixed-point operations and 16-bit multiplies. Both support a shuffle or permute operation on each of 32 source bytes, although the Vector/SIMD Multimedia Extensions also support several merge and replicate operations. Both support many rotates and shifts, but there are about twice as many choices among the SPU instructions.

Some details of their distinctive features are given below. For summary lists of the instructions, see *Section A.3* on page 737 and *Section B.1* on page 761.

23.1.3.1 *Vector/SIMD Multimedia Extension Instructions*

The Vector/SIMD Multimedia Extensions provide these operations that are not directly supported by the SPU instruction set:

- Saturating math.
- Sum-across.
- Log_2 and 2^x.
- Ceiling and floor.
- Complete byte instructions.

Vector/SIMD Multimedia Extensions support several vector-pack and unpack operations, although comparable operation can be performed with SPU instructions. Vector/SIMD Multimedia Extension compare instructions have an optional recording form that is useful for creating conditional-branch tests, as described in *Section 23.1.1.1* on page 670.

The first implementation of the *CBEA* (the CBE processor) supports instructions with a graphics rounding mode. This mode allows programs written with Vector/SIMD Multimedia Extension instructions to produce floating-point results that are equivalent in precision to those written in the SPU instruction set. In this mode, as in the SPU environment, the default rounding mode is r*ound to zero*, denormals are treated as zero, and there are no infinities or NaNs. For details, see *Section A.3.3* on page 742.

Mechanisms are available in both the PPE and SPE to hide memory-load latency. The PPE PowerPC instructions include data-cache block touch instructions that can cause data to be prefetched into the cache, plus cache-line locking instructions to prevent prefetched data from being cast out. However, the advantages of prefetching data are limited by the small number of registers with which to operate on the data. The SPEs, in contrast, support DMA transfers in parallel with computation, and their 128 registers provide ample resources with which to operate on data. Thus, SPU loads may be better suited for deterministic real-time applications.

23.1.3.2 *SPU Instructions*

The SPU instruction set provides these operations that are not directly supported by a single Vector/SIMD Multimedia Extension instruction:

- Immediate operands.
- Double-precision floating-point.
- Sum of absolute difference.
- Count ones in bytes.
- Count leading zeros.
- Equivalence.
- AND complement.
- OR complement.
- Extend sign.
- Gather bits.
- Form select mask.
- Integer multiply and accumulate.
- Multiply subtract.
- Multiply float.
- Shuffle byte special conditions.
- Carry and borrow generate.
- Sum bytes across.
- Extended shift range.

SPU single-precision floating-point instructions execute at about twice the speed of Vector/SIMD Multimedia Extension single-precision floating-point, and the SPU instructions—unlike the Vector/SIMD Multimedia Extension instructions—also support double-precision vector operations.

The SPU ISA defines compare instructions for setting masks. These masks can be used in three-operand select instructions to create efficient conditional assignments to avoid difficult-to-predict branches. The SPU also supports hint for branch instructions (*Section 24.3.3* on page 691) that avoid branch penalties on taken branches when the branch can be predicted sufficiently early.

23.2 Porting SIMD Code from the PPE to the SPEs

It is sometimes easier to write SIMD programs by writing them first for the PPE than porting them to the SPEs. This approach postpones some SPE-related considerations—such as the size of the SPE's LS, data movements, and debug—until after the port. This approach can also allow partitioning of the work into simpler steps on the SPEs. Nevertheless, it is important to begin considering code partitioning for multiple SPUs early in the development of program architecture.

After Vector/SIMD Multimedia Extension code is working properly on the PPE, a strategy for parallelizing the algorithm across multiple SPEs can be implemented. This is often, but not always, a data-partitioning method. The effort might involve converting from Vector/SIMD Multimedia Extension intrinsics to SPU intrinsics, adding data-transfer and synchronization constructs, and tuning for performance. It might be useful to test the impact of various techniques, such as DMA double-buffering, loop unrolling, branch elimination, alternative intrinsics, number of SPEs, and so forth (see *Section 24* on page 681 for some examples). Debugging tools such as the static timing-analysis tool and the IBM Full System Simulator for the Cell Broadband Engine are available to assist this effort.

Alternatively, experienced CBE programmers may prefer to skip the Vector/SIMD Multimedia Extension coding phase and go directly to SPU programming. In some cases, SIMD programming can be easier on an SPE than the PPE because of the SPE's unified register file.

The earlier chapters in this tutorial describe the Vector/SIMD Multimedia Extension and SPU programming environments and some of their differences. Armed with knowledge of these differences, one can devise a strategy for developing code that is portable between the PPE and the SPEs. The strategy one should employ depends upon the type of instructions to be executed, the variety of vector data types, and the performance objectives. Solutions span the range of simple macro translation to full functional mapping.

23.2.1 Code-Mapping Considerations

There are several challenges associated with mapping code designed for one instruction set and compiled for another instruction set. These including performance, unmappable constructs, limited size of LS, and equivalent precision, as described below.

23.2.1.1 *Performance*

Simplistic mapping of low-level intrinsics may not deliver the full performance potential of the SPE, depending upon the intrinsics used. Understanding the dynamic range of the mapping's operands—such as whether the range of operands is known to be restricted—can reduce the performance impact of simple mapping.

For example, consider mapping the **spu_shuffle** intrinsics (**shufb** instruction) to **vec_perm** (**vperm** instruction). If it was known than the shuffle-pattern bytes are all in the range 0x'0' through 0x'1F', then **spu_shuffle** can be directly mapped to **vec_perm**. Otherwise, the mapping requires several instructions.

23.2.1.2 *Unmappable Constructs*

Differences in the processing of intrinsics make simple translation of certain intrinsics unmappable. The unmappable SPU intrinsics include:

- `stop` and `stopd`.
- conditional halt.
- interrupt enable and disable.
- move to and from status control and special purpose registers.
- channel instructions.

- branch on external data.

23.2.1.3 *Limited Size of LS*

Vector/SIMD Multimedia Extension programs mapped to SPU programs might not fit within the LS of the SPE, either because the program is initially too big or because mapping expands the code.

23.2.1.4 *Equivalent Precision*

The SPU instruction set does not fully implement the IEEE 754 single-precision floating-point standard[55] (default rounding mode is r*ound to zero*, denormals are treated as zero, and there are no infinities or NaNs). Therefore, floating-point results on an SPE may differ slightly from floating-point results using the PPE's PowerPC instruction set. In addition, all estimation intrinsics (for example, ceiling, floor, reciprocal estimate, reciprocal square root estimate, exponent estimate, and log estimate) do not have equivalent accuracy on the SPU and PPE PowerPC instruction sets.

However, the instructions in the PPE's Vector/SIMD Multimedia Extension have a *graphics rounding mode* (enabled by default) that is not available in the standard PowerPC Architecture. This mode is a special feature of the *Cell Broadband Engine Architecture*. It allows programs written with Vector/SIMD Multimedia Extension instructions to produce floating-point results for add, subtract, multiply, multiply-add, and multiply-subtract that are equivalent in precision to those written in the SPU instruction set (but that differ from the estimate instructions). In this Vector/SIMD Multimedia Extension graphics rounding mode, as in the SPU environment, the default rounding mode is r*ound to zero*, denormals are treated as zero, and there are no infinities or NaNs. For details on the graphics rounding mode, see *Appendix A.3.3 Graphics Rounding Mode* on page 742.

23.2.2 Simple Macro Translation

For many programs, it is possible to use a simple macro translation strategy for developing code that is portable between the Vector/SIMD Multimedia Extension and SPU instruction sets. The keys to simple macro translation are described in the sections that follow.

23.2.2.1 *Use a Compatible Vector-Literal Construction Format*

The Vector/SIMD Multimedia Extension and SPU instruction sets support two styles of constructing literal vectors: curly brace (preferred style for SPE) and parenthesis (deprecated AltiVec style). Some compilers support both styles. A set of construction macros can be used to insulate programs from any differences in the tools.

There are multiple methods of constructing literal vectors, and all tools may not support the same construction format. The key differences in the methods are syntax (the deprecated AltiVec *parenthesis style* or the preferred *curly-braces style*) and handling of unspecified elements. The AltiVec style requires either a single element or all elements be specified. If only a single element

55. The Vector/SIMD Multimedia Extension instructions also do not fully implement the IEEE standard. The instruction sets are architected this way so as to better support applications such as graphics and real-time code. For details, see the *PowerPC Microprocessor Family: Vector/SIMD Multimedia Extension Technolgy Programming Environments Manual*.

is specified, then all elements of the vector are initialized with the specified value. The curly braces style allows one to specify any number of vector elements. The Vector/SIMD Multimedia Extension and SPU instruction sets and most compilers support both syntax styles. Unspecified elements are cleared to ‘0’.

A simple set of construction macros can be used to insulate programs from any differences in the tools and assure portability of vector construction. For example:

```
#ifdef ALTIVEC_STYLE
    /* deprecated legacy support */
    #define VECTOR_UINT(a,b,c,d) \
        ((vector unsigned int)(a,b,c,d))
#else // CURLY_BRACE_STYLE
    /* preferred CBEA format */
    #define VECTOR_UINT(a,b,c,d) \
        ((vector unsigned int){a,b,c,d})
#endif
```

23.2.2.2 *Use Single-Token Vector Data Types*

The *SPU C/C++ Language Extensions* document specifies a set of single-token vector data types, listed in *Table B-7* on page 773. Because these are single-token, the data types can be easily redefined by a preprocessor to the desired target processor. Additional single-token data types have been standardized for the unique Vector/SIMD Multimedia Extension data types. *Table 23-3* lists the additional data types. Also, see *Table 23-2* on page 671.

Table 23-3. Additional Vector/SIMD Multimedia Extension Single-Token Data Types

Vector Data Type	Single-Token Data Type
vector bool char	vec_bchar16
vector bool short	vec_bshort8
vector bool int	vec_bint4
vector pixel	vec_pixel8

23.2.2.3 *Use Intrinsics that Map One-to-One*

Regardless of the technique used to provide portability, performance will be optimized if the operations map one-to-one between Vector/SIMD Multimedia Extension intrinsics and SPU intrinsics. The SPU intrinsics that map one-to-one with Vector/SIMD Multimedia Extension (except specific intrinsics) are shown in *Table 23-4*. The Vector/SIMD Multimedia Extension intrinsics that map one-to-one with SPU are shown in *Table 23-5*.

Table 23-4. SPU Intrinsics with One-to-One Vector/SIMD Multimedia Extension Mapping

SPU Intrinsic	Vector/SIMD Multimedia Extension Intrinsic	For Data Types
spu_add	vec_add	vector operands only, no scalar operands
spu_genc	vec_addc	all
spu_and	vec_and	vector operands only, no scalar operands
spu_andc	vec_andc	all
spu_avg	vec_avg	all
spu_cmpeq	vec_cmpeq	vector operands only, no scalar operands
spu_cmpgt	vec_cmpgt	vector operands only, no scalar operands
spu_convtf	vec_ctf	limited scale range (5 bits)
spu_convts	vec_cts	limited scale range (5 bits)
spu_convtu	vec_ctu	limited scale range (5 bits)
spu_madd	vec_madd	float only
spu_mulhh	vec_mule	all
spu_mulo	vec_mulo	halfword vector operands only, no scalar operands
spu_nmsub	vec_nmsub	float only
spu_nor	vec_nor	all
spu_or	vec_or	vector operands only, no scalar operands
spu_re	vec_re	all
spu_rl	vec_rl	vector operands only, no scalar operands
spu_rsqrte	vec_rsqrte	all
spu_sel	vec_sel	all
spu_sub	vec_sub	vector operands only, no scalar operands
spu_genb	vec_subc	vector operands only, no scalar operands
spu_xor	vec_xor	vector operands only, no scalar operands

Table 23-5. Vector/SIMD Multimedia Extension Intrinsics with One-to-One SPU Mapping (Page 1 of 2)

Vector/SIMD Multimedia Extension Intrinsic	SPU Intrinsic	For Data Types
vec_add	spu_add	halfwords, words, and floats only (not bytes)
vec_addc	spu_genc	all
vec_and	spu_and	all
vec_andc	spu_andc	all
vec_avg	spu_avg	unsigned chars only
vec_cmpeq	spu_cmpeq	all
vec_cmpgt	spu_cmpgt	all
vec_ctf	spu_convtf	all

Table 23-5. Vector/SIMD Multimedia Extension Intrinsics with One-to-One SPU Mapping (Page 2 of 2)

Vector/SIMD Multimedia Extension Intrinsic	SPU Intrinsic	For Data Types
vec_cts	spu_convts	all
vec_ctu	spu_convtu	all
vec_madd	spu_madd	all
vec_mulo	spu_mulo	halfwords only (not bytes)
vec_nmsub	spu_nmsub	all
vec_nor	spu_nor	all
vec_or	spu_or	all
vec_re	spu_re	all
vec_rl	spu_rl	halfwords and words only (not bytes)
vec_rsqrte	spu_rsqrte	all
vec_sel	spu_sel	all
vec_sub	spu_sub	halfwords, words, and floats only
vec_subc	spu_genb	all
vec_xor	spu_xor	all

23.2.3 Full Functional Mapping

Because intrinsics have function-call syntax, code written for a specific functional unit can be mapped using inline functions. The keys to full functional mapping are:

- *C++ Function Overloading*—Since generic-intrinsic mappings are polymorphic (that is, one generic intrinsic maps to one of several instructions), C++ function overloading must be used to assure correct mapping.
- *Map Non-Supported Data Types to Unique Data Types*—Using C++ function overloading to map a generic intrinsic to a specific function requires that its parameter types be unique for every specific function. The Vector/SIMD Multimedia Extensions support vector data types that are not valid on an SPU, and vice-versa. Therefore, these data types must be mapped to a supported type on the target processor element. *Table 23-6* on page 678 recommends a mapping that guarantees that the overloading function will uniquely select the correct mapped function without having to mandate the new data types be supported.

Table 23-6. Unique SPU-to-Vector/SIMD Multimedia Extension Data-Type Mappings

SPU Data Type	Vector/SIMD Multimedia Extension Data Type
vector unsigned long long	vector bool char
vector signed long long	vector bool short
vector double	vector bool int
vector unsigned short	vector pixel[1]

1. There are insufficient unique vector types on the SPU to provide a unique mapping for the *vector pixel* data type. This causes an overloading conflict when attempting to support the vec_unpackl and vec_unpackh intrinsics for both *vector signed short/vector bool short* and *vector pixel* types. Generic mapping routines must use compile This conflict can be resolved by using conditional compilation selection of the desired mapping. Simultaneous mapping in a single object is not possible without further language extensions.

- *Use Intrinsics that Map One-to-One*—Use intrinsics that map one-to-one, as described in *Section 23.2.2.3* on page 676.

23.2.4 Code-Portability Typedefs

To improve code portability between PPE and SPU programs, single-token typedefs can be provided for the vector keyword data types. An example of such typedefs is shown in *Table B-7* on page 773. Because they are single-token, they serve as class names for extending generic intrinsics for mapping to-and-from Vector/SIMD Multimedia Extension intrinsics and SPU intrinsics.

23.2.5 Compiler-Target Definition

To support the development of code that can be conditionally compiled for multiple targets, such as SPU and PPU, the *SPU C/C++ Language Extensions* specify that compilers must define `__SPU__` when code is being compiled for the SPU. As an example, the following code supports misaligned quadword loads on both the SPU and PPU. The `__SPU__` define is used to conditionally select which code to use. The code that is selected will be different depending on the processor target.

```
vector unsigned char load_qword_unaligned(vector unsigned char *ptr)
{
    vector unsigned char qw0, qw1, qw;
#ifdef __SPU__
    unsigned int shift;
#endif
    qw0 = *ptr;
    qw1 = *(ptr+1);
#ifdef __SPU__
    shift = (unsigned int)(ptr) & 15;
    qw = spu_or(spu_slqwbyte(qw0, shift, spu_rlmaskqwbyte(qw1, (signed)(shift - 16)));
#else /* PPU */
    qw = vec_perm(qw0, qw1, vec_lvsl(0, ptr));
#endif
    return (qw);
}
```

24. SPE Programming Tips

This chapter provides general advice and code examples about how to write efficient code for the SPE. Specific suggestions and examples are given for overlapping DMA and computation, eliminating and predicting branches, loop unrolling and pipelining, vectorizing a loop, and transformations and table-lookups. References to more detailed descriptions of these and other topics are also given.

Among the most general guidelines for SPU coding are these:

- *Intrinsics*—Use intrinsics to achieve machine-level control without the need to write assembly-language code. Understand how the intrinsics map to assembly instructions and what the assembly instructions do. See *Section B* on page 761 for a summary of the instruction set and its C-language intrinsics, and see the *SPU C/C++ Language Extensions* document for complete details.
- *Overlap DMA Transfers with Computation*—Data transfers and computation can proceed in parallel by using multibuffering techniques. In addition, use the SPEs rather than the PPE to initiate most DMA transfers. See *Section 24.1* on page 681.
- *Fetch and Issue Rules*—An understanding of the fetch and issue rules will maximize your dual-issue rate. See *Section 24.2* on page 687.
- *Design for Limited Local Store*—Each LS is limited to 256 KB for program, stack, local data structures, and DMA buffers. Many optimization techniques put pressure on this limited resource. As such, all optimizations may not be possible for a given application. Often it is possible to reduce LS pressure by dynamically managing the program store using code overlays.
- *Branch Instructions*—Eliminate branches or reduce branch misprediction using hint-for branch instructions. See *Section 24.3* on page 689.
- *Loop Optimization*—Unroll and pipeline loops. See *Section 24.4* on page 698.
- *Integer Multiplication*—Use the SPU's 16 x16 bit multiplier to best advantage by avoiding 32-bit integer multiplies. See *Section 24.7* on page 705.
- *Table Lookups and Permutations*—Use the shuffle-bytes instruction for efficient table lookups and reorganization of operands. See *Section 24.6* on page 701.
- *Scalar Code*—Scalar (sub-quadword) loads and stores require multiple instructions and have long latencies. *Section 24.8* on page 705 has advice about efficient use of scalar code.

For a general overview of the SPE architecture, see *Section 3* on page 63.

24.1 DMA Transfers

This section builds on the basic description of DMA and DMA list transfer methods given in *Section 19.4* on page 521. We describe here how to overlap DMA transfers with computation by using double-buffering and multi-buffering, plus techniques to improve DMA performance.

24.1.1 Initiating DMA Transfers from SPEs

Performance is usually best if you initiate DMA transfers from the SPEs that need the transfers rather than from the PPE. There are eight SPEs and only one PPE. The PPE can have only four in-flight data references (plus two for L2-cache loads and stores). Each of the eight SPEs, in contrast, can have up to 16 in-flight DMA transfers. Because of these considerations, the SPEs can initiate many more DMA transfers than the PPE in a given period of time.

24.1.2 Overlapping DMA Transfers and Computation

In order to hide data-access latencies, use double- or multi-buffering techniques. You can either double-buffer the data (typical) or double-buffer the code, depending upon the code and data sizes and data-access patterns. This section outlines basic strategies for overlapping DMA transfers with computation on an SPU.

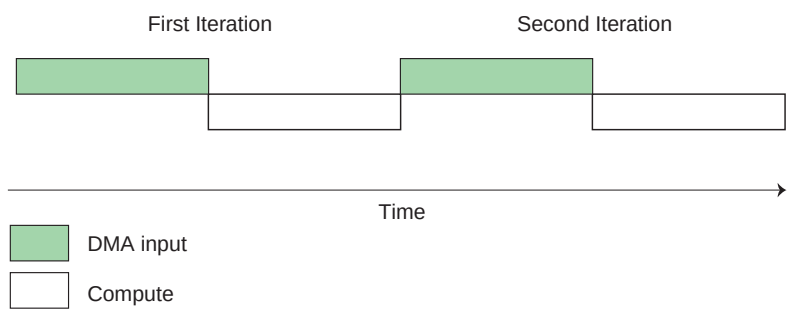
24.1.2.1 *Double Buffering*

Consider a simple SPU program that performs the following steps:

1. DMA incoming data from effective address (EA) in main storage to LS buffer B.
2. Wait for the transfer to complete.
3. Compute on data in buffer B.
4. Repeat as necessary.

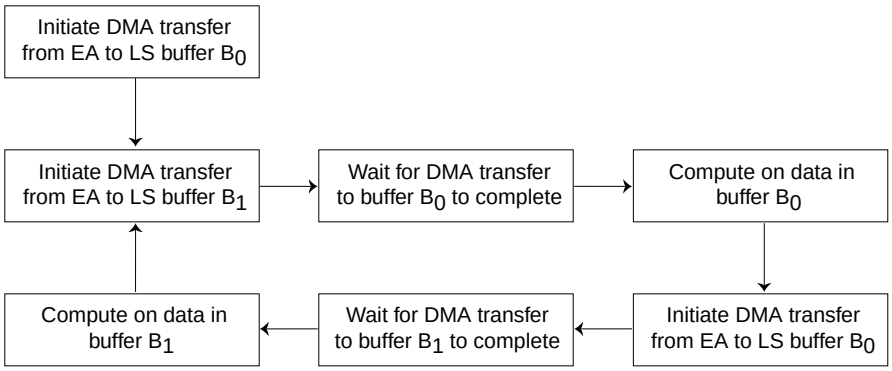
This sequence has no overlap between data transfer and computation. If considered over time, the time graph might look like *Figure 24-1*.

Figure 24-1. Serial Computation and Transfer



When these steps are known to iterate more than once, performance may be improved by allocating two LS buffers, B_0 and B_1, and overlapping computation on one buffer with data transfer for the next. This technique is known as *double-buffering*. *Figure 24-2* shows a flow diagram for this double buffering scheme. Double buffering is a form of *multibuffering,* which is the method of using multiple buffers in a circular queue to overlap computation and data transfer.

Figure 24-2. DMA Transfers Using a Double-Buffering Method



The double-buffering sequence is:

1. Initiate DMA transfer of incoming data from EA to LS buffer B_0.
2. Initiate DMA transfer of incoming data from EA to LS buffer B_1.
3. Wait for transfer of buffer B_0 to complete.
4. Compute on data in buffer B_0.
5. Initiate DMA transfer of incoming data from EA to LS buffer B_0.
6. Wait for transfer of buffer B_1 to complete.
7. Compute on data in buffer B_1.
8. Repeat steps 2 through 7 as necessary.

As the new time graph in *Figure 24-3* shows, computation and data transfers proceed concurrently once the first DMA for B_0 completes.

Figure 24-3. Parallel Computation and Transfer

Double-buffering can be achieved on the SPU by applying tag-group identifiers (*Section 19.2.5* on page 511). Tag-group ID 0 is applied to all transfers involving B_0 (steps 1, 3, and 5) and tag-group ID 1 is applied to all transfers involving B_1 (steps 2 and 6).

The key to double-buffering on the SPU lies in properly setting the tag-group mask, such that the program only waits for the particular buffer of interest. To wait for B_0, software sets the tag-group mask to include only tag ID 0 and requests conditional tag-status update (step 3). To wait for B_1, software sets the tag-group mask to include only tag ID 1 and again requests conditional tag-status update (step 6).

The following C-language program fragment shows how to initiate a buffer transfer.

```
#include <spu_intrinsics.h>
#include <spu_mfcio.h>

volatile void *B[2]; /* Pointers to LS Buffers */

/* Initiate transfer using LS buffer B[idx] */
static inline void xfer(unsigned int ea, unsigned int size, unsigned int id)
{
    spu_mfcdma32(B[idx], ea, size, idx, MFC_GET_CMD);
}
```

The following C-language program fragment shows how to wait for a buffer transfer to complete.

```
/* Wait for B[idx] transfer to complete. */
static inline void wait_xfer(unsigned int idx)
{
    unsigned int tag_mask = (1 << idx);

    spu_writech(MFC_WrTagMask, tag_mask);
    spu_mfcstat(MFC_TAG_UPDATE_ALL);
}
```

Both incoming or outgoing data transfers can cause bottlenecks for application performance. It may therefore be necessary to implement buffering schemes for both input and output.

The following C-language program fragment shows one way to implement double-buffering. The code starts the first DMA transfer, then enters the loop. The loop starts the next transfer and waits on the first one. When the first completes, the code calls the `use_data()` function to process the fresh buffer of DMA data. The code then toggles the buffer index and loops to start the next transfer. The code waits for the oldest transfer to complete, and then calls `use_data()` to process the fresh data. The process repeats until all the data has been transferred and processed.

We assume the `MFC_` macro names have been defined previously.

```
/* Example C code demonstrating double buffering using buffers B[0] and B[1].
 * In this example, an array of data starting at the effective address
 * eahi|ealow is DMAed into the SPU's local store in 4 KB chunks and processed
 * by the use_data subroutine.
 */

#include <spu_intrinsics.h>
#include <spu_mfcio.h>
```

```
#define BUFFER_SIZE 4096
volatile unsigned char B[2][BUFFER_SIZE] __attribute__ ((aligned(128)));

void double_buffer_example(unsigned int ea, int buffers)
{
    int next_idx, idx = 0;

    // Initiate first DMA transfer
    spu_mfcdma32(B[idx], ea, BUFFER_SIZE, idx, MFC_GET_CMD);

    ea += BUFFER_SIZE;
    while (--buffers) {
        next_idx = idx ^ 1;
        spu_mfcdma32(B[next_idx], ea, BUFFER_SIZE, idx, MFC_GET_CMD);
        ea += BUFFER_SIZE;
        spu_writech(MFC_WrTagMask, 1 << idx);
        (void)spu_mfcstat(MFC_TAG_UPDATE_ALL); // Wait for previous transfer done
        use_data(B[idx]);                      // Use the previous data
        idx = next_idx;
    }
    spu_writech(MFC_WrTagMask, 1 << idx);
    (void)spu_mfcstat(MFC_TAG_UPDATE_ALL);    // Wait for last transfer done
    use_data(B[idx]);                         // Use the last data
}
```

24.1.2.2 *Multibuffering*

Data buffering can be extended to use more than two buffers, provided that enough space is available in the LS. If more than two buffers are involved, the technique is known as multi-buffering.

Building on the concepts from the previous section, a basic recipe for multibuffered data transfers on the SPU is:

1. Allocate multiple LS buffers, $B_0..B_n$.
2. Initiate transfers for buffers $B_0..B_n$. For each buffer B_i, apply tag group identifier i to transfers involving that buffer.
3. Beginning with B_0 and moving through each of the buffers in round robin fashion:
   - Set tag group mask to include only tag i, and request conditional tag status update.
   - Compute on B_i.
   - Initiate the next transfer on B_i.

This algorithm waits for and processes each B_i in round-robin order, regardless of when the transfers complete with respect to one another. In this regard, the algorithm uses a strongly ordered transfer model. Strongly ordered transfers are useful when the data must be processed in a known order.

An alternative would be to set the tag-group mask to include all outstanding buffer transfers, and then request conditional tag-status update for any tags that have completed. The buffers could be processed when their transfers complete, rather than in the order that their transfers were initiated. This would represent a weakly ordered transfer model. Weakly ordered transfers may be useful when the data can be processed in any order.

24.1.2.3 *Shared I/O Buffers*

This section examines a special case of multibuffering in which the storage for the incoming buffer is reused for subsequent outbound transfers. This allows the program to compute on input data from a buffer, store the results back to that same buffer, and then initiate an outbound transfer to write the results back to main storage. The advantage of using shared buffers is that LS memory requirements can be reduced.

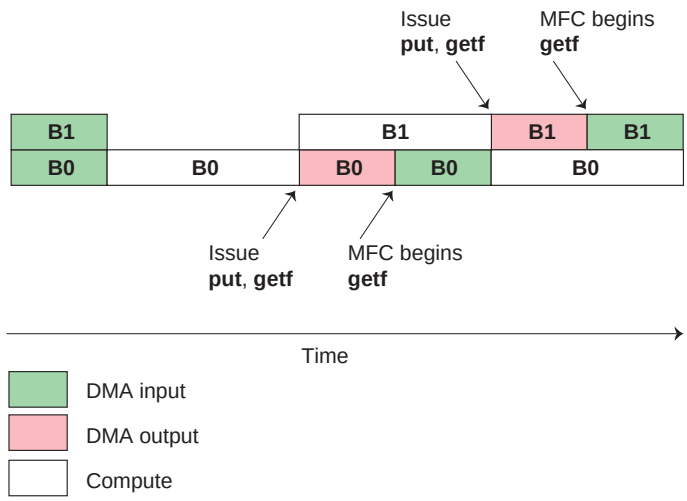
Shared I/O buffers may be particularly useful when a task has roughly a one-to-one correspondence between the amount of incoming and outgoing data. Vertex transformation (or more generally vertex shading) is an example of a task in which there is a one-to-one correspondence between incoming and outgoing data. Image convolution is another example.

However, an ordering dependency is introduced when sharing buffers for both input and output: previous outbound transfers need to complete before subsequent incoming transfers can be initiated on the same buffer. Such an ordering dependency can lead to increased program complexity, particularly when multiple incoming and outgoing transfers are in flight.

The MFC supports fenced variants of all the **get** and **put** commands (*Section 20.1.4.4* on page 564). The fence attribute causes a command to be locally ordered with respect to all previously issued commands within the same tag group. This is useful for shared buffers.

Figure 24-4 show how a fenced **get** command works. The ordering provided by the fence ensures that the **get** is not initiated until the **put** from the same buffer has completed. By using fenced **get** commands, an SPU program does not need to wait for outbound transfers to complete. Both parallelism and program simplicity can be maintained.

*Figure 24-4. Shared I/O Buffers with Fenced **get** Command*



24.1.2.4 *Guidelines for Multibuffering Data*

To use multibuffering effectively for data, follow these rules for DMA transfers on the SPE:

- Use multiple LS buffers.
- Use unique DMA tag IDs, one for each LS buffer.
- Use fenced or barriered commands to order the DMA transfers within a tag group.
- Use separate barrier commands to order DMA transfers within the MFC command queue.
- Keep array elements a power-of-2 size to avoid multiplication when indexing.

24.1.3 DMA Transfers and LS Accesses

When using DMA buffers, declare the DMA buffers as volatile to ensure that buffers are not accessed by SPU load or store instructions until after DMA transfers have completed.

The *SPU C/C++ Language Extensions* specify that channel commands are ordered with respect to volatile-memory accesses. The DMA commands specify LS addresses as volatile, void pointers. By declaring all DMA buffers as volatile, it forces all accesses to these buffers to be performed (that is, they cannot be cached in a register) and ordered.

Depending on the algorithm and buffer access, this can result in additional memory accesses (loads and stores) and additional constraints on the compiler's optimizing scheduler that need not exist. The solution to this problem is to either:

- Load DMA-buffer contents into non-volatile (typically local stack) variables, and reference these variables.
- Declare the DMA buffers as non-volatile, and perform DMA transfers and synchronization in non-inlined functions. This orders the DMA transfers and memory accesses by separating the two by a function-call boundary.

When coding DMA transfers, exploit DMA list transfers whenever possible. See *Section 19.2.2 DMA List Commands* on page 509 for details. See *Section 3.1.1.3* on page 66 for LS access priorities.

24.2 SPU Pipelines and Dual-Issue Rules

The SPU has two pipelines, even (pipeline 0) and odd (pipeline 1), into which it can issue and complete up to two instructions per cycle, one in each of the pipelines. Whether an instruction goes to the even or odd pipeline depends on its instruction type, which is related to the execution unit that performs the function. Each execution unit is assigned to one of the two pipelines.

To obtain high performance from the pipeline:

- Design for balanced pipeline use. Typically, the algorithm dictates the instruction mix. However, there may be multiple ways to achieve the same computational results. Choosing the one that achieves balanced pipeline use will often result in improved performance.
- Unroll loops and interleave computation to hide latency (reduce dependency stalls) and improve dual-issue rates.

Table 24-1 and *Table 24-2* provide an overview of:

- *Pipeline*—The pipeline instructions are issued on.
- *Stalls*—The number of additional cycles before another instruction of the same type can be issued. For example, double-precision floating-point operations have a 6 cycle stall. Therefore, for back to back double-precision floating-point operations, the second operation will be issued at least 7 cycles after the first operation.
- *Latency*—The number of instructions before the result is available.

Table 24-1. Pipeline 0 Instructions and Latencies

Pipeline-0 Instructions	Latency (clocks)	Stall (clocks)
Single-precision floating-point operations	6	0
Integer multiplies, convert between floating-point and integer, interpolate	7	0
Immediate loads, logical operations, integer add and subtract, signed extend, count leading zeros, select bits, carry and borrow generate	2	0
Double-precision floating-point operations	7	6
Element rotates and shifts	4	0
Byte operations (count ones, absolute difference, average, sum)	4	0

Table 24-2. Pipeline 1 Instructions and Latencies

Pipeline-1 Instructions	Latency (clocks)	Stall (clocks)
Shuffle bytes, quadword rotates, and shifts	4	0
Gather, mask, generate insertion control	4	0
Estimate	4	0
Loads, stores, branch hints	6	0
Branches	4	0
Channel operations, move to/from SPRs	6	0

The SPU issues all instructions in program order according to the pipeline assignment. Each instruction is part of a doubleword-aligned instruction-pair called a *fetch group*. A fetch group can have one or two valid instructions. This means that the first instruction in the fetch group is from an even word address, and the second instruction from an odd word address. The SPU processes fetch groups one at a time, continuing to the next fetch group when the current fetch group becomes empty. An instruction becomes issueable when register dependencies are satisfied and there is no structural hazard (resource conflict) with prior instructions or LS contention due to DMA or ECC activity. See *Section 3.1.1.3* on page 66 for LS access priorities.

Dual-issue occurs when a fetch group has two issueable instructions in which the first instruction can be executed on the even pipeline and the second instruction can be executed on the odd pipeline. If a fetch group cannot be dual-issued, but the first instruction can be issued, the first instruction is issued to the proper execution pipeline and the second instruction is held until it can be issued. A new fetch group is loaded after both instructions of the current fetch group are issued.

For details on the SPU fetch and issue rules, see *Section B.1.3* on page 768.

24.3 Eliminating and Predicting Branches

The SPU hardware assumes sequential instruction flow. A branch instruction has the potential of disrupting the assumed sequential flow. Correctly predicted branches execute in one cycle, but a mispredicted branch (conditional or unconditional) incurs a penalty of 18 to 19 cycles, depending on the address of the branch target. Considering the typical SPU instruction latency of two-to-seven cycles, mispredicted branches can seriously degrade program performance. The branch instructions also restrict a compiler's ability to optimally schedule instructions by creating a barrier on instruction reordering.

The most effective method of reducing the impact of branches is to eliminate them using three primary methods—inlining, unrolling, and predication. The second-most effective method of reducing the impact of branches is to use the hint-for branch instructions. If software speculates that the instruction branches to a target path, a branch hint is provided. If a hint is not provided, software speculates that the branch is not taken (that is, instruction execution continues sequentially).

24.3.1 Function-Inlining and Loop-Unrolling

Function-inlining and loop-unrolling are two techniques often used to increase the size of *basic blocks* (sequences of consecutive instructions without branches), which increases scheduling opportunities.

Function-inlining eliminates the two branches associated with function-call linkage. These include the *branch and set link* (such as **brasl**) for function-call entry, and the *branch indirect* (such as **bi**) for function-call return. Loop-unrolling eliminates branches by decreasing the number of loop iterations. Loop unrolling can be manual, programmer-directed, or compiler-automated. Typically, branches associated with looping are inexpensive because they are highly predictable. However, if a loop can be fully unrolled, then all branches can be eliminated—including the final nonpredicted branch. For more about loop unrolling, see *Section 24.4* on page 698.

Over-aggressive use of inlining and loop unrolling can result in code that reduces the LS space available for data storage or, in the extreme case, is too large to fit in the LS.

24.3.2 Predication Using Select-Bits Instruction

The *select-bits* (**selb**) instruction is the key to eliminating branches for simple control-flow statements (for example, *if* and *if-then-else* constructs). An *if-then-else* statement can be made branchless by computing the results of both the *then* and *else* clauses and using *select bits* (**selb**) to choose the result as a function of the conditional. If computing both results costs less than a mispredicted branch, then there are additional savings.

For example, consider the following simple if-then-else statement:

```
unsigned int a, b, d;
. . .
if (a > b) d += a;
else       d += 1;
```

This code sequence when directly converted to an SPU instruction sequence without branch optimizations would look like:

```
   clgt   cc, a, b
   brz    cc, else
then:
   a      d, d, a
   br     done
else:
   ai     d, d, 1
done:
```

Using the *select bits* instruction, this simple conditional becomes:

```
   clgt   cc, a, b                  /* compute the greater-than condition */
   a      d_plus_a, d, a            /* add d + a */
   ai     d_plus_1, d, 1            /* add d + 1 */
   selb   d, d_plus_1, d_plus_a, cc /* select proper result */
```

This example shows:

- Both branches were eliminated, and the correct result was placed in `d`.
- New registers were needed to maintain potential values of `d` (`d_plus_a` and `d_plus_1`). This does not put significant pressure on the register file since the register file is so large and life of these variables is very short.
- The rewritten code sequence is smaller.
- The latency of the operations permits the scheduler to cover most of the cost of computing both conditions. Further scheduling these instructions with those before and after this code sequence will likely improve performance even further.

Here is an example of using the select bits with C intrinsics. This code fragment uses the `spu_cmpgt`, `spu_add`, and `spu_sel` intrinsics to eliminate conditional branches.

```
vector unsigned int va, vb, vd;

va = spu_promote(a, 0);
vb = spu_promote(b, 0);
vd = spu_promote(d, 0);

vd = spu_sel(spu_add(vd, 1), spu_add(vd, va), spu_cmpgt(a, b));

d = spu_extract(vd, 0);
```

Note: *Most optimizing compilers will exploit the* ***selb*** *instructions for small conditionals.*

24.3.3 Branch Hints

General-purpose processors have typically addressed branch prediction by supporting hardware look-asides with branch history tables (BHT), branch-target address caches (BTAC), or branch-target instruction caches (BTIC). The SPU supports branch prediction through a set of *hint for branch* (HBR) instructions (**hbr**, **hbra**, and **hbrr**) and a branch-target buffer (BTB). These instructions support efficient branch processing by allowing programs to avoid the penalty of taken branches.

Although the effects of hint-for branch instructions are not defined by the *Synergistic Processor Unit Instruction Set Architecture*, the hint-for branch instructions provide three kinds of advance knowledge about future branches:

- Address of the branch target (that is, where will the branch take the flow of control).
- Address of the actual branch instruction (known as the *hint-trigger address*).
- Prefetch schedule (when to initiate prefetching instructions at the branch target).

Hint-for branch instructions have no program-visible effects. They provide a hint to the SPU about a future branch instruction, with the intention that the information be used to improve performance by prefetching the branch target.

If software provides a branch hint, software is speculating that the instruction branches to the branch target. If a hint is not provided, software speculates that the instruction does not branch to a new location (that is, it stays inline). If speculation is incorrect, the speculated branch is flushed and refetched. It is possible to sequence multiple hints in advance of multiple branches. As with all programmer-provided hints, care must be exercised when using branch hints because, if the information provided is incorrect, performance might degrade.

24.3.3.1 *Hint-for Branch Instructions*

The SPE hint-for branch instructions are shown in *Table 24-3*. There are immediate and indirect forms for this instruction class. The location of the branch is always specified by an immediate operand in the instruction.

Table 24-3. Hint-for Branch Instructions

Instruction	Description
hbr s11, ra	Hint for branch (r-form). Hint that the instruction addressed by the sum of the address of the current instruction and the signed extended 11-bit value, s11, will branch to the address contained in word element 0 of register ra. This form is used to hint function returns, pointer function calls, and other situations that give rise to indirect branches.
hbra s11, s18	Hint for branch (a-form). Hint that the instruction addressed by the sum of the address of the current instruction and the signed extended 11-bit value, s11, will branch to the address specified by the sign extended, 18-bit value s18.
hbrr s11, s18	Hint for branch relative. Hint that the instruction addressed by the sum of the address of the current instruction and the signed extended 11-bit value, s11, will branch to the address specified by the sum of the address of the current instruction and sign extended, 18-bit value s18.

24.3.3.2 *Hint-for Branch Fetch*

Hint-for branch instructions load the SPU's branch-target buffer (BTB) with a branch target address and an address of a branch instruction (the hint-trigger address). After loading, the BTB monitors the instruction stream as it goes into the issue stage of the pipeline. When the address of the instruction going into issue matches the hint trigger address, the hint is triggered, and the SPU speculates to the target address.

Branch hints arbitrate for the LS as though they were loads, except that they require three idle DMA cycles before they can issue. See *Section 3.1.1.3* on page 66 for details about LS access priority.

24.3.3.3 *Hint Trigger*

The BTB is invalid at SPU program start. After a branch hint is loaded, the hint remains valid either until it is replaced by another hint or is invalidated by a **sync** instruction. Hints are disabled for the first instruction pair sent to issue after a flush.

If software's objective is to load the hint early enough so that the target is issued in the cycle after the branch, the branch hint should precede the branch by at least eleven cycles plus four instruction-pairs.

24.3.3.4 *Hint Stall and Pipelined Hint Mode*

It is expected that most hints are used for end-of-loop branches or paired with an upcoming branch. It is also possible to sequence multiple hint in advance of multiple branches in a pipeline fashion.

After the BTB and the trigger are loaded, they remain available for triggering until either another hint is loaded or a synchronizing event occurs (**sync** or start). In particular, the BTB is not cleared if an instruction-sequencing error[56] occurs. Therefore, it may be appropriate to hoist hint instructions from certain simple loops into loop-initialization code. The following types of loops are probably not candidates for hint hoisting:

- Loops that execute in interrupt-enabled mode.
- Loops that contain **sync** instructions.
- Loops that feature a likely-taken branch in the loop body.

If possible, a hint should precede the branch by at least eleven cycles plus four instruction pairs. More separation between the hint and branch will probably improve the performance of applications on future SPU implementations. Hints that are too close to the branch do not affect the speculation after the branch. Hint stall has been added to reduce the number of instructions required between the hint and the branch, so that the hinted target follows the branch through issue. If there are at least four instruction pairs between the hint instruction and the branch instruction (based on the branch-address index in the hint instruction), the SPU enters hint stall. Hint stall does not stall the hint; rather, it holds the branch instruction in or before the stage of the

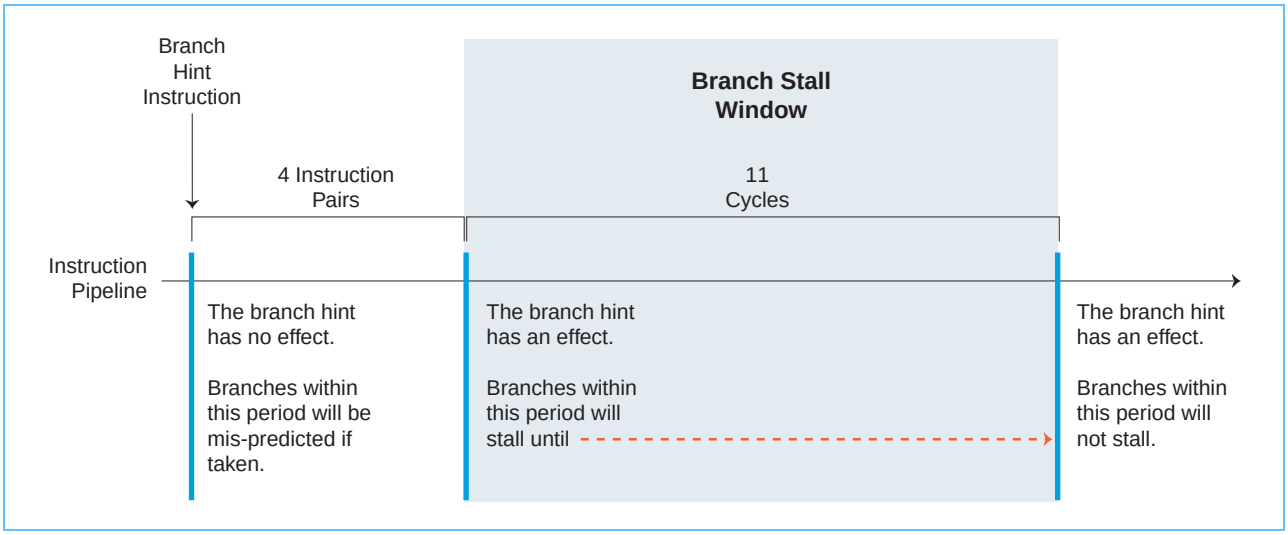
56. An instruction-sequencing error occurs when instructions are executed in a different order other than expected. The SPU expects instructions to be executed sequentially unless modified by hint trigger.

pipeline in which triggering occurs. The branch is held there until the hint trigger is loaded. When the hint is ready for trigger, hint stall is released, proper address-matching occurs, and the target follows the branch through the issue pipeline.

Hint stall is set up by hints whose branches are at least eight, but less than 32, instructions ahead. These hints are said to be stallable. Issue stalls if the hinted branch advances to or just before the stage of the pipeline in which triggering occurs, and the BTB has not yet been loaded by the hint instruction.

Figure 24-5 shows the branch stall window with respect to when branch hints have an effect.

Figure 24-5. Branch Stall Window



If another stallable hint is issued in the window between the first stallable hint and the issuance of the original hinted branch, the SPU disables the hint stall and enters pipelined hint mode. When pipelined hint mode is entered, the hint stall is negated and no further hint stall is generated for the duration of the pipelined hint mode. Pipelined hint mode lasts for 16 cycles after the last stallable hint is issued in pipelined hint mode. Hints executed during pipelined hint mode are written to the hint buffer in order, and are available for trigger from the time they are written until the time the next hint is written. Performance enhancements obtained by using pipelined hint mode may not be portable to future implementations of SPU.

Four instruction pairs and one cycle must separate the hint from the branch in order for the branch to be predicted to the taken path. If the hint is closer than this to the branch, hint stall does not occur. To use hint stall, the instruction sequence must be padded with no operation (**nop**) and no operation-load (**lnop**) instructions. If hint stall occurs, the speculation may still be incorrect. In this case, the incorrect speculation is flushed when it gets to branch resolution.

24.3.3.5 ***Rules for Using Branch Hints***

The following general rules apply to the *hint for branch* (HBR) instructions:

- An HBR instruction should be placed at least 11 cycles followed by four instruction pairs before the branch instructions being hinted by the HBR instruction. In other words, an HBR instruction must be followed by at least 11 cycles of instructions, followed by eight instruc-

tions aligned on an even address boundary. More separation between the hint and branch will probably improve the performance of applications on future SPU implementations.

- If an HBR instruction is placed too close to the branch, then a hint stall will result. This results in the branch instruction stalling until the timing requirement of the HBR instruction is satisfied.
- If an HBR instruction is placed closer to the hint-trigger address than four instruction pairs plus one cycle, then the hint stall does not occur and the HBR is not used.
- Only one HBR instruction can be active at a time. Issuing another HBR cancels the current one.
- An HBR instruction can be moved outside of a loop and will be effective on each loop iteration as long as another HBR or **sync** instruction is not executed.
- The HBR instruction must be placed within 64 instructions of the branch instruction.
- The HBR instruction only affects performance.

The HBR instructions can be used to support multiple strategies of branch prediction. These include:

- *Static Branch Prediction*—Prediction based upon branch type or displacement (*Section 24.3.4 Program-Based Branch Prediction* on page 694), and prediction based upon profiling or linguistic hints (*Section 24.3.5 Profile or Linguistic Branch-Prediction* on page 696).
- *Dynamic Branch Prediction*—Software caching of branch-target addresses (*Section 24.3.6 Software Branch-Target Address Cache* on page 697), and using control flow to record branching history (*Section 24.3.7 Using Control Flow to Record Branch History* on page 697).

24.3.4 Program-Based Branch Prediction

Statically predicting branches based upon the program can improve branch prediction. Ball and Larus (1993) developed a set of heuristics that performed well for a large and diverse set of programs. Even though these programs are not necessarily appropriate candidates for SPU execution, the heuristic are still valid.

Table 24-4 proposes static branch predictions for various types of program constructs. For example, “always” means that the branch should always be predicted taken, and “never” means that branch should never be predicted taken.

Table 24-4. Heuristics for Static Branch Prediction (Page 1 of 2)

Program Feature	Static Prediction
Unconditional branch	always
Conditional branch	never
Loop-iteration branch	always
Loop-termination branch	never
Call and return	always
Pointer comparison	false
Less than zero	false

Table 24-4. Heuristics for Static Branch Prediction (Page 2 of 2)

Program Feature	Static Prediction
Greater than zero	true
Floating-point equal	false
Blocks containing store instructions[1]	avoid
Path length[1]	shortest
Short-circuit evaluation	avoid

1. "Blocks containing store instructions" and "Path length" refer to the properties of code in the predicted branch target.

For the path-length heuristic in *Table 24-4*, if prediction is correct, branch penalty is small, but if prediction is incorrect, the mispredict branch penalty is small compared to the computation cost of the executed path.

For the short-circuit evaluation heuristic in *Table 24-4*, a common optimization performed by compilers on complex conditionals is to terminate (short-circuit) the evaluation as soon as possible. Consider the following example:

```
C Source Conditional:
    if ((a > b) && (c > d))
Assembly:
    clgt   cc_ab, a, b
    clgt   cc_cd, c, d
    and    cc, cc_ab, cc_cd
    brz    cc, not_true
        . . .
not_true:
```

Short-circuit evaluation would introduce a branch on the first condition, so the second need not be evaluated:

```
Assembly w/ Short-Circuit Evaluation:
    clgt   cc_ab, a, b
    brz    cc_ab, not_true
    clgt   cc_cd, c, d
    brz    cc_cd, not_true
        . . .
not_true:
```

This technique is not well-suited for effective branch hints on the SPU because there is generally not adequate runway for hinting all the branches. As such, this optimization technique should probably be avoided.

24.3.5 Profile or Linguistic Branch-Prediction

A common approach to generating static branch prediction is to use expert knowledge that is obtained either by feedback-directed optimization techniques or using linguistic hints supplied by the programmer.

There are many arguments against profiling large bodies of code, but most SPU code is not like that. SPU code tends to be well-understood loops. Thus, obtaining realistic profile data should not be time-consuming. Compilers should be able to use this information to arrange code so as to increase the number of fall-through branches (that is, conditional branches not taken). The information can also be used to select candidates for loop unrolling and other optimizations that tend to unduly consume LS space.

Programmer-directed hints can also be used effectively to encourage compilers to insert optimally predicted branches. Even though there is some anecdotal evidence that programmers do not use them very often, and when they do use them, the result is wrong, this is likely not the case for SPU programmers. SPU programmers generally know a great deal about performance and will be highly motivated to generate optimal code.

The *SPU C/C++ Language Extension* specification defines a compiler directive mechanism for branch prediction (*Section B.2.5 Compiler Directives* on page 779). The `__builtin_expect` directive allows programmers to predicate conditional program statements. The following example demonstrates how a programmer can predict that a conditional statement is false (a is not larger than b).

```
if(__builtin_expect((a>b),0))
  c += a;
else
  d += 1;
```

Not only can the `__builtin_expect` directive be used for static branch prediction, it can also be used for dynamic branch prediction. The return value of `__builtin_expect` is the value of the `exp` argument, which must be an integral expression. For dynamic prediction, the `value` argument can be either a compile-time constant or a variable. The `__builtin_expect` function assumes that `exp` equals `value`.

A static-prediction example might look like this:

```
if (__builtin_expect(x, 0)) {
    foo(); /* programmer doesn't expect foo to be called */
}
```

A dynamic-prediction example might look like this:

```
cond2 = .../* predict a value for cond1 */
...
cond1 = ...
if (__builtin_expect(cond1, cond2)) {
    foo();
}
```

```
cond2 = cond1;/* predict that next branch is the same as the previous */
```

Compilers may require limiting the complexity of the expression argument because multiple branches could be generated. When this situation occurs, the compiler must issue a warning if the program's branch expectations are ignored.

24.3.6 Software Branch-Target Address Cache

The HBR instructions can also be used to implement a software version of a branch-target address cache (BTAC). Consider the following example in which a loop contains a conditional branch:

```
   lqr    i, iterations
loop:
    . . .
   brz    cc, skip      // condition variable cc
   . . .
skip:
   ai     i, i, -1      // deccrement loop counter
   brnz   i, loop
```

The conditional branch (**brz** instruction, above) could be predicted as follows:

```
   lqr    i, iterations
   ila    hint, skip    // initial hint - skip condition
loop:
   hbr    br_addr, hint // hint based upon previous iteration
    . . .
   ila    hint, skip    // assume hint for next iteration
br_addr:
   brz    cc, skip
   ila    hint, br_addr+4// correct hint for next iteration
   . . .
skip:
   ai     i, i, -1
   brnz   i, loop
```

In this example, the `hint` variable is used to store the previous iteration's branch target, under the assumption that the previous iteration's branch result is a good predictor of the next iteration's branch. It is possible to extend this concept and keep a history of branch results to be used as the predictor.

24.3.7 Using Control Flow to Record Branch History

Another technique for improving branch prediction is to replicate loops with its branches reversed and taken branches being vectored to the other copy of the loop. For example, consider the following if-then clause within a loop:

```
     op1
loop:
     op2
    brz    cc, label1
     op3
label1:
     op4
    brnz   cntr, loop
     op5
```

This loop can be rewritten into two tightly coupled loops as follows:

```
LOOP 1 (branch not taken)    LOOP 2 (branch taken)
-------------------------    ---------------------

     op1
loop:                        loop2:
     op2                          op2
    brz    cc, label3            brnz   cc, label2
label2:                      label3:
     op3                          op4
label1:                          brnz   cntr, loop2
     op4                         br     label4
    brnz   cntr, loop
label4:
```

This optimization is effectively a 1-bit branch predictor for the conditional. The left loop (LOOP 1) is used when predicting the conditional as true. The right loop (LOOP 2) is used when predicting the conditional as not true. The main problem with this technique is that code can grow significantly as it is applied to more complex and larger code sequences. Judicious application of this technique is recommended.

24.4 Loop Unrolling and Pipelining

Loops are the foundation in nearly all programs, especially streaming applications. If the number of loop iterations is a small constant, then consider removing the loop altogether. Otherwise, consider unrolling the loop if the loop is relatively independent (that is, an iteration of the loop is not dependent upon the previous iteration). The SPU has a large register file and significant loop unrolling can be accomplished before register spill occurs. Register spilling occurs when the instantaneous number of active variables exceeds the size of the register file. Unrolled loops provide additional computation for the optimizer to improve issue rates and reduce dependency stalls.

When unrolling loops, additional local variables may be required to eliminate false dependencies amongst the loop variables. Failure to eliminate these false dependencies can prevent unrolled loops from being interleaved by the compiler.

Note: *Applications using auto-vectorization technology should not explicitly remove, unroll, or pipeline loops. This only adds complexity to the auto-vectorization process.*

Consider a sample workload, called `xformlight`, that performs basic graphics vertex-processing, four vertices at a time. The vertex processing includes 4 x 4 vertex transformation, perspective division, one local light computation with OpenGL-style specularity, color conversion, clamping, and RGBA color packing. *Table 24-5* shows how loop unrolling affects the performance of this workload. These values are typical of many computational workloads.

Table 24-5. Loop Unrolling the `xformlight` Workload

Metric	Unroll Factor			
	1	2	4	8
Normalized Performance	1.00	1.52	1.73	1.66
Cycles Per Instruction (CPI)	1.35	0.91	0.76	0.67
Dual Issue Rate	3.3%	19.8%	34.3%	35.8%
Dependency Stalls	27.2%	5.9%	0.9%	1.5%
Registers Used	78	103	128	128
Text Size (bytes)	768	1344	3076	5252

Most loops have the same basic structure, as shown in *Figure 24-6*. Per iteration they load input data, perform computation, and finally store the results. Since loads, stores, quadword rotates, and shuffles execute on pipeline 1, and most computation instructions execute on pipeline 0, loops can be software pipelined to improve dual-issue rates by computing at the same time as loading and storing data. *Figure 24-7* shows how software pipelining functions, and *Table 24-5* shows some representative results of software pipelining the unrolled `xformlight` workload.

Figure 24-6. Basic Loop

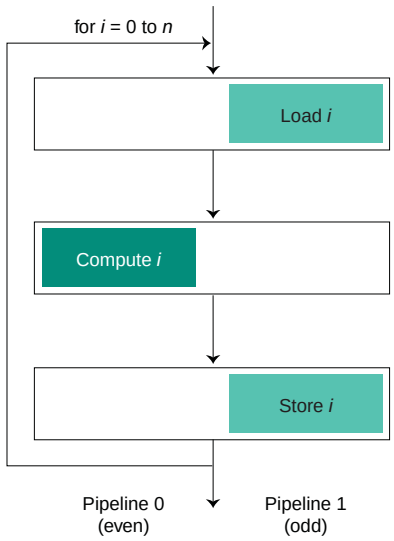


Figure 24-7. Software-Pipelined Loop

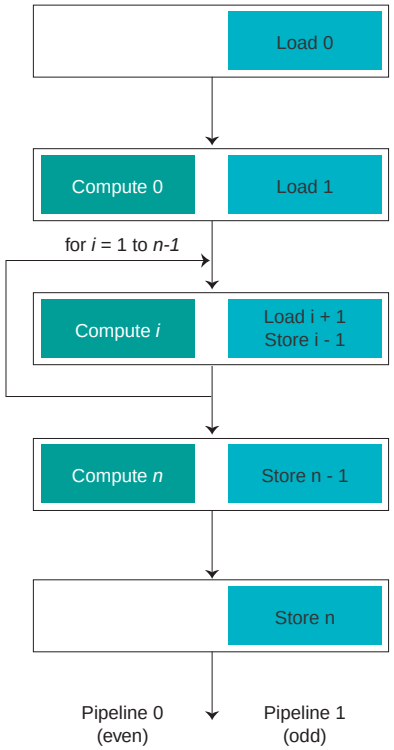


Table 24-6. Software Pipelining the Unrolled `xformlight` Workload

Metric	Unroll Factor	
	2	4
Normalized Performance	1.82	1.83
Cycles Per Instruction (CPI)	0.75	0.69
Dual Issue Rate	36.7%	47.8%
Dependency Stalls	1.3%	0.5%
Resource Conflicts	1.0%	0.5%
Registers Used	114	128
Text Size (bytes)	2436	3468
Speedup vs. non-pipelined	1.16	1.08

24.5 Offset Pointers

The PPE's PowerPC instruction set supports *load/store with update* instructions. These instructions allow one to sequentially index through an array without the need of additional instructions to increment the array pointer. The SPU does not support this instruction form. Instead, one should exploit the SPU d-form load instructions by specifying small, literal array offsets from the base array pointer.

For example, consider the following PPE code that exploits the PowerPC *store with update* instruction:

```
#define FILL_VEC_FLOAT(_q, _data)   *(vector float)(_q++) = _data;

FILL_VEC_FLOAT(q, x);
FILL_VEC_FLOAT(q, y);
FILL_VEC_FLOAT(q, z);
FILL_VEC_FLOAT(q, w);
```

The same code can be modified for SPU execution as follows:

```
#define FILL_VEC_FLOAT(_q, _offset, _data) *(vector float)(_q+(_offset)) = _data;

FILL_VEC_FLOAT(q, 0, x);
FILL_VEC_FLOAT(q, 1, y);
FILL_VEC_FLOAT(q, 2, z);
FILL_VEC_FLOAT(q, 3, w);
q += 4;
```

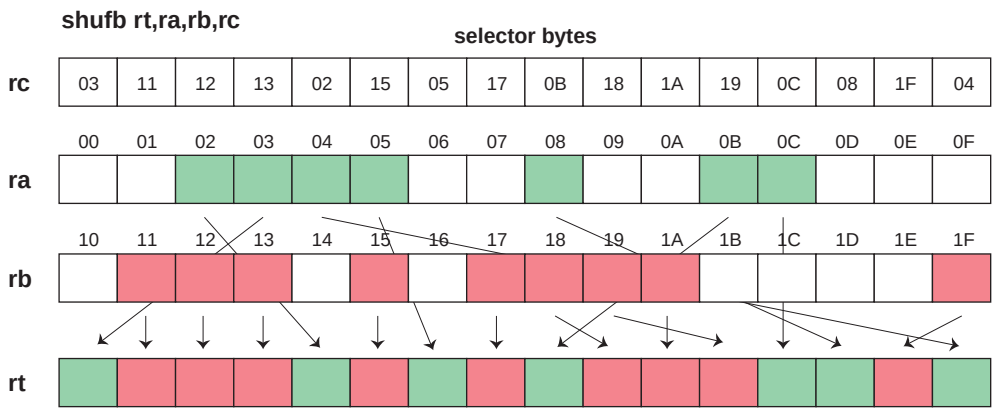
24.6 Transformations and Table-Lookups

24.6.1 The Shuffle-Bytes Instruction

The SPU's **shufb** (shuffle bytes) instruction, which is comparable to the **vperm** instruction for the PPE, supports shuffles and permutations that can be used for many purposes, including table lookups, transformations between array-of-structure (AOS) and structure-of-array (SOA) vector forms, and operand alignment such as endian-reversal.

Figure 24-8 shows an example of how the **shufb** (shuffle bytes) instruction works. Each of the vectors, **ra**, **rb**, **rc**, and **rt**, contain sixteen byte elements. Each byte of **rc** is used to select a byte from either **ra**, **rb**, or a constant (0, 0x'80', or 0x'FF'). For example, an **rc** byte of 0x'03' will select byte element 3 of **ra**, and an **rc** byte of 0x'15' will select byte element 5 of **rb**. The results are placed in the corresponding bytes of **rt**.

Figure 24-8. Shuffle Bytes Instruction



A compiler can use the **shufb** instruction to perform a (sub-quadword) scalar store, using a shuffle pattern generated by one of the *Generate Controls for Insertion* instructions (**cbd**, **cbx**, **cdd**, **cdx**, **chd**, **chx**, **cwd**, **cwx**).

24.6.2 Fast SIMD 8-Bit Table Lookups

Many applications require table lookups. An example that uses 8-bit table lookups is the Advanced Encryption Standard (AES) byte-substitution (*ByteSub)* transformations. See the *Advanced Encryption Standard (AES)*, Federal Information Processing Standards Publications, FIPS PUB 197, 2001, and http://csrc.nist.gov/CryptoToolkit/aes/rijndael/.

Here is an example of an AES ByteSub transformation:

```
unsigned char s_box[256];
unsigned char state[16];
for (i=0; i<16; i++) {
    state[i] = s_box[state[i]];
}
```

Because table lookups involve an indirection, most vector processors do not support SIMD table lookups. Instead, programmers often must fall back to scalar operations, which lack the acceleration found in parallel operations.

24.6.2.1 *Scalar Table Lookup*

Assuming an AES state block is stored in a 128-bit quadword vector and the `s_box` array is either a 256-entry byte array of unknown alignment or a 16-entry quadword array, a scalar SPU implementation of the AES ByteSub transformation could be implemented as follows:

```
   ila     s_box, s_box_address    # initialize s_box array pointer
   il      cnt, 16                 # initialize loop counter
next_byte:                         # for each byte in the state quadword
```

```
andi    idx, state, 0xFF        #  strip off other bytes in the word
lqx     new_state, s_box, idx   # fetch sbox array quadword indexed by state byte
cbx     insert, s_box, idx      # generate shuffle pattern for next instruction
shufb   state, new_state, state, insert  # insert sbox lookup into state quadword
ai      cnt, cnt, -1            #  decrement loop counter
rotqbyi state, state, 1         #  rotate state quadword for next byte
brnz    cnt, next_byte          #  branch if more bytes in the quadword
```

The above example uses approximately seven instructions per byte-lookup. By completely unrolling the loop, the number of instructions per byte-lookup can be reduced to five. Similar results can be obtained using the PPE's Vector/SIMD Multimedia Extension unit.

24.6.2.2 *Exploiting the Shuffle-Bytes Operation*

The key to improving the performance of 8-bit table lookups on an SPE is the **shufb** instruction, described in *Section 24.6.1* on page 701. This instruction is capable of performing 16 simultaneous 8-bit table lookups in a 32-entry table. Assuming the table is quadword-aligned, the following SPU instructions perform 16 simultaneous table lookups in a 32-entry byte table:

```
lqa    table0_15, table + 0
lqa    table16_31, table + 16
andbi  idx, state, 0x1F
shufb  state, table0_15, table16_31, idx
```

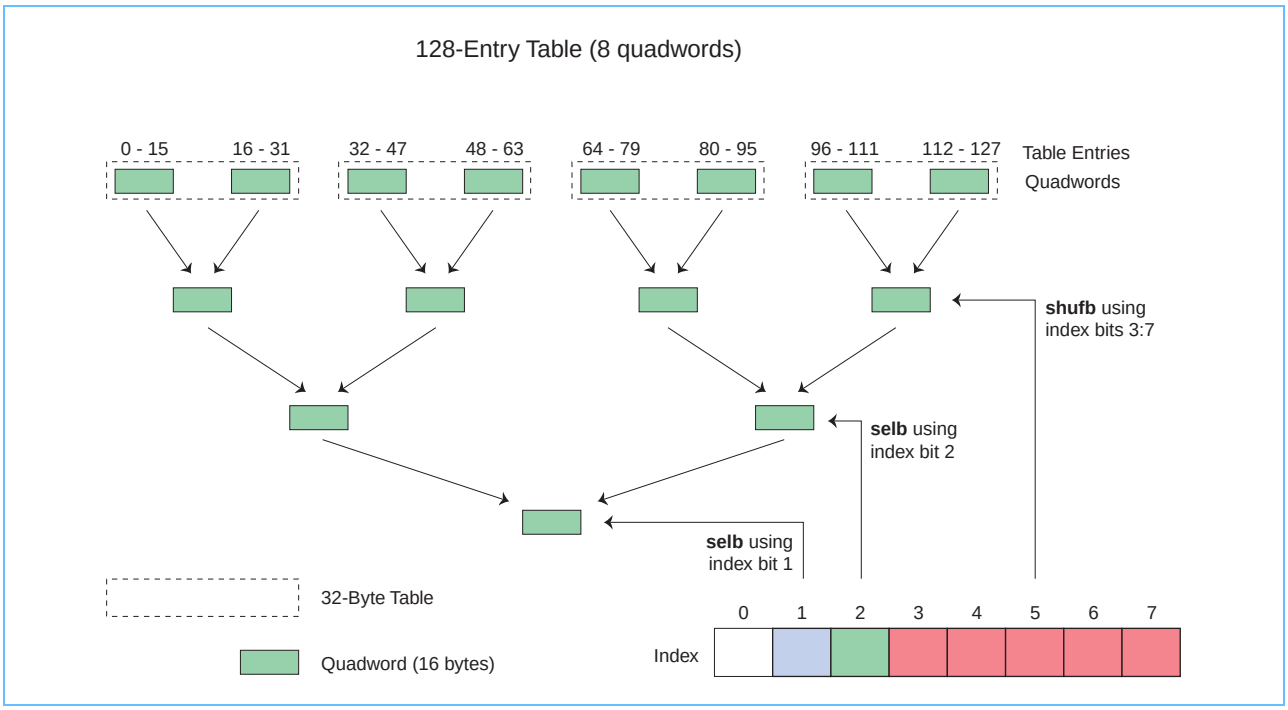
Most real-life examples require tables larger than 32 entries. Such larger tables can be handled by performing a binary-tree pruning process on a series of 32-entry table lookups. The pruning is performed using successively more significant bits in the table index.

For example, *Figure 24-9* shows the operations required to perform a 128-entry table lookup. In this case, the 128 entries of the table are loaded, one quadword at a time, into eight quadwords (quadwords 0 through 7). The five least-significant bits are used as the shuffle pattern to perform a byte-for-byte select from the four 32-entry subtables. Each successive bit of the table index are used as a selector (using the **selb** instruction, *Section 24.3.2* on page 689) to choose the correct subtable value. This is repeated for each of the remaining most-significant bits of the table index, as follows:

```
vector unsigned char idx;
vector unsigned char tbl[8];
vector unsigned char tbl0_1, tbl2_3, tbl4_5;
vector unsigned char tbl6_7, tbl0_3, tbl4_7;
vector unsigned char lsb3_7, bit3, bit2;
vector unsigned char result;
lsb3_7 = spu_and(idx, 0x1F);
tbl0_1 = spu_shuffle(tbl[0],tbl[1],lsb3_7);
tbl2_3 = spu_shuffle(tbl[2],tbl[3],lsb3_7)
tbl4_5 = spu_shuffle(tbl[4],tbl[5],lsb3_7);
tbl6_7 = spu_shuffle(tbl[6],tbl[7],lsb3_7);
```

```
bit2 = spu_cmpeq(spu_and(idx,0x20),0x20);
tbl0_3 = spu_sel(tbl0_1,tbl2_3,bit2);
tbl4_7 = spu_sel(tbl4_5,tbl6_7,bit2);
bit1 = spu_cmpgt(idx,0x3F);
result = spu_sel(tbl0_3,tbl4_7,bit1);
```

Figure 24-9. 128-Entry Table Lookup Example



24.6.2.3 *Performance Characterization*

The previous sections have compared a scalar implementation of a table lookup with a SIMD implementation. Both implementations use quadword data-loads and loop-unrolling to maximize dual-issue rates and minimize dependencies. The performance results are shown in *Table 24-7*. The results clearly show the performance benefit of SIMD table lookups. Despite the fact that dual-issue efficiency is compromised in the SIMD version, the SIMD results show performance gains of 3.7 to 14.3 times that of scalar lookups, and code reductions of 4.6 to 27 times that of scalar lookups. The scalar implementation has higher dual-issue rates because the number of instructions and instruction mix provides more opportunities to achieve higher dual-issue rates. This example demonstrates that small CPI does not always corollate to good performance.

Table 24-7. Performance Comparison (Page 1 of 2)

Technique	Table Size	Code Size[1]	Cycles/Instruction	Cycles/Byte Looked Up
scalar	1 to 255	1760	0.536	3.58
SIMD	1 to 32	64	1.07	0.250

Table 24-7. Performance Comparison (Page 2 of 2)

Technique	Table Size	Code Size[1]	Cycles/Instruction	Cycles/Byte Looked Up
SIMD	1 to 64	128	0.774	0.375
SIMD	1 to 128	224	0.691	0.594
SIMD	1 to 255	384	0.653	0.969

1. Inner-loop instructions only, expressed in bytes.

24.7 Integer Multiplies

The SPU contains only a 16 x16 bit multiplier. Therefore, to perform a 32-bit integer multiply, it takes five instructions—three 16-bit multiplies and two adds to accumulate the partial products.

To avoid extraneous multiply cycles, observe the following rules:

- Make array-element size a power of 2. This avoids multiplication when indexing.
- If the operands are less than 16 bits in size, cast them to `unsigned short` prior to multiplication. This takes maximum advantage of the multiplier.
- Always cast constants, because they have an implicit type of `int`.
- To avoid inadvertent introduction of signed extends and masks due to casting, consider introducing a macro to explicitly perform an integer multiply whose operands are 16-bits or less. For example:

```
#define MULTIPLY(a, b)\
    (spu_extract(spu_mulo((vector unsigned short)spu_promote(a,0),\
    (vector unsigned short)spu_promote(b, 0)),0))
```

24.8 Scalar Code

24.8.1 Scalar Loads and Stores

The SPU only loads and stores a quadword at a time. As such, scalar (sub-quadword) loads and stores require multiple instructions and have long latencies. This is due to the fact that scalar loads must be rotated into the preferred scalar slot (*Figure 3-6* on page 75), and stores require a read-modify-write operation to insert the scalar into the quadword being stored. This overhead is demonstrated by the following scalar load and store sample:

```
void add1(int *p) { *p += 1; }

ASSEMBLY

add1:
    lqd        $4, 0(p)         # load quadword containing int pointed to by p
    rotqby $5, $4, p            # rotate int into preferred scalar slot
    ai         $5, $5, 1        # add 1 to int
```

```
cwd       $6, 0(p)         # generate shuffle pattern for scalar word insertion
shufb     $4, $5, $4, $6   # insert scalar into quadword
stqd      $4, 0(p)         # store updated quadword back to local store
```

There are several strategies for making scalar code (code that is not appropriate for vectorization) more efficient. These include:

- Change the scalars to quadword vectors. This may seem wasteful, however, if you consider the 3 extra instructions associated with loading and storing scalars, this trade-off can actually have a positive impact on code size.
- Cluster scalars into groups and load multiple scalars at a time using a quadword memory access. Manually extract or insert the scalars on an as-needed basis. This will eliminate redundant loads and stores.

These strategies have been applied to an implementation of RC4, a character base encryption algorithm that uses a 256-byte dynamic state table. The basic algorithm is:

```
/* cipher msg_in to msg_out */
unsigned char idx1, idx2, *msg_in, *msg_out;
...
for (i=0; i<msg_len; i++)  {
    idx2 = state[++idx1] + idx2;
    SWAP(state[idx1], state[idx2]);
    msg_out[i] = msg_in[i] ^ (state[idx1] + state[idx2])
}
```

Because each iteration of the algorithm is dependent upon previous iteration, it cannot be parallelized. However, by exploiting the strategies outlined above, significant performance enhancements can still be achieved, as shown in *Table 24-8*. The 256-byte state table is expanded into a 256-quadword state table in which all 16 elements of the unsigned character vector contain the same byte value. The input and output messages are loaded and stored a quadword at a time to eliminate the extraneous loads and stores and their respective latencies. These changes resulted in an 86% performance speedup.

Table 24-8. RC4 Scalar Improvements

Implementation	Instructions	Cycles	CPI	speedup
original	540120	723245	1.34	-
optimized	265794	388457	1.46	1.86

24.8.2 Promoting Scalar Data Types to Vector Data Types

The **spu_promote** and **spu_extract** SPU intrinsics are provided to efficiently promote scalars to vectors, or vectors to scalars. These intrinsics are listed in *Table 24-9*. When promoting a scalar operand to a vector operand, be sure to promote the scalar to the *preferred scalar slot* (*Figure 3-6* on page 75) to avoid the need for additional instructions. The preferred scalar slot for byte, halfword, word, and doubleword operations is shown in *Table 24-10*.

Table 24-9. Intrinsics for Changing Scalar and Vector Data Types

Instruction	Description
d = spu_promote(a, element)	Promote a scalar to a vector.
d = spu_extract(a, element)	Extract a vector element from its vector.

Table 24-10. Preferred Scalar Slot

Scalar Data Type	Byte Address of Preferred Scalar Slot
Byte	3
Halfword	1
Word	0
Doubleword	0

24.9 Unaligned Loads

On the SPE, shift-left and shift-right instructions must be used to perform an unaligned quadword vector load:

```
vector unsigned char load_vec_unaligned(unsigned char *ptr)
{
    vector unsigned char qw0, qw1, qw;
    unsigned int shift;

    shift = (unsigned int)(ptr) & 15;
    qw0 = *((vector unsigned char *)(ptr));
    qw1 = *((vector unsigned char *)(ptr+16));
    qw = spu_or(spu_slqwbyte(qw0, shift),
               spu_rlmaskqwbyte(qw1, (signed int)(shift-16)));
    return (qw);
}
```

In contrast, unaligned loads using the PPE's Vector/SIMD Multimedia Extensions use the load-left (**lvlx**) and load-right (**lvrx**) instructions. Unaligned stores use the store-left (**stvlx**) and store-right (**stvrx**) instructions. Two examples are:

```
# Rb contains a pointer to the misaligned quadword.
# Vt contains the loaded quadword vector.
lvlx                Vhi, 0, Rb            # load left-most portion
addi                Rb, Rb, 16
lvrx                Vlo, 0, Rb            # load right-most portion
vor                 Vt, Vhi, Vlo          # combine the two portions

# Rb contains a pointer to the misaligned quadword.
# Vs contains the quadword to be stored.
stvlx                Vs, 0, Rb            # store the left-most portion
```

```
        addi            Rb, Rb, 16
        stvrx           Vs, 0, Rb           # store the right-most portion
```

Appendices, Glossary, Index, and Revision Log

This section includes:

- *Appendix A PPE Instruction Set and Intrinsics* on page 711.
- *Appendix B SPU Instruction Set and Intrinsics* on page 761.
- *Appendix C Performance Monitor Signals* on page 781.
- *Glossary* on page 823.
- *Index* on page 859.
- *Revision Log* on page 875.

Appendix A. PPE Instruction Set and Intrinsics

A.1 PowerPC Instruction Set

An overview of the PPE's PowerPC instructions and data types is given in *Section 2.4* on page 51. The sections that follow summarize key points of the instructions. For a complete description, see the *PowerPC Architecture* documents.

A.1.1 Data Types

Section 2.4.1 on page 51 contains a list of the PowerPC data types.

A.1.2 Instructions

Table A-1 on page 712 summarizes the PPE instruction-performance characteristics. The entries in this table are organized by the functional units that execute the instructions. Some considerations for interpreting the table:

- An instruction can be executed in one or more execution units, as shown in *Table A-1* on page 712.
- All instructions are pipelined and—except for double-precision floating-point instructions, which stall for 6 cycles—can issue in a single clock cycle if there are no dependencies between consecutive loads.
- Latency is the issue-to-issue latency for a dependent instruction. For example, if an **add** issues in cycle 20 and the soonest a dependent **xor** could issue is cycle 22, the latency for **add** is listed as 2. There are multiple latencies for some instructions when the instruction writes more than one type of register. In general, latency will be one cycle larger than use penalty.
- Throughput refers to the maximum sustained rate at which the PPE can execute instructions of the type noted in the absence of any dependencies and assuming infinite caches. It is shown as instructions per cycle (IPC).
- Some fields are labelled "N/A" to indicate that a more elaborate description is required (and is best understood by taking a broader view of the machine as a whole), This includes complex microcoded instructions. Instructions that do not modify a register are also labelled "N/A."
- In the Notes column, "MC" indicates instructions that are microcoded. "CSI" indicates a context-synchronizing instruction.
- The Execution Units column indicates BRU: Branch unit, IU: Instruction unit, FPU: Floating-point unit, FXU: Fixed-point unit, and LSU: Load and store unit.

Table A-1. PowerPC Instructions, by Execution Unit (Page 1 of 10)

Mnemonic	Description	Execution Unit[1]	Latency (cycles)[2]	Throughput (IPC)	Notes[3]	Comments
b ba bl bla	Branch	BRU	See Note 2	1 per latency cycle		Unconditional branches always “predicted” correctly.
bc bca bcl bcla	Branch conditional	BRU	1 (lr / ctr)	1 per latency cycle		Branch direction predicted at the top of the pipeline. May update the link stack.
bclr bclrl	Branch conditional to link register	BRU	1 (lr / ctr)	1 per latency cycle		Branch direction and target address predicted at the top of the pipeline. May use the link stack.
bcctr bcctrl	Branch conditional to count register	BRU	1 2 (ctr) (lr / ctr)	1 per latency cycle		Branch direction and target address predicted at the top of the pipeline. May use the link stack.
crand cror crxor crnand crnor crnandc creqv crorc	CR logical operations	BRU	1	1 per latency cycle		
mcrf	Move CR field	BRU	1	1 per latency cycle		
mtspr(lr) **mtspr(ctr)**	Move to LR Move to CTR	BRU	1	1 per latency cycle		
mcrxr	Move to CR from XER	BRU	1	1 per latency cycle		
mfcr	Move from CR	BRU	2	1 per 3 latency cycles		Bit 11 = 0
mfcr		BRU	~34	~1 per 34 latency cycles		Bit 11 = 1. This form is nonpipelined.
sc rfi rfid	System call Return from interrupt	IU	See Note 2	See Note 2	CSI	
lbz lbzx lhz lhzx lwz lwzx ld ldx	Load	LSU	2	1 per latency cycle		5-cycle latency for **load** followed by dependant **load**.
lhbrx lwbrx	Load byte-reversed	LSU	2	1 per latency cycle		5-cycle latency for **load** followed by dependant **load**.
lmw	Load multiple word	LSU	See *Table A-4* on page 726	1 per latency cycle	MC	Microcode generates inline sequence of basic loads.
lswi (naturally aligned)	Load string word immediate	LSU	See *Table A-4* on page 726	1 per latency cycle	MC	Microcode assumes natural alignment and generates inline sequence of basic loads.

Table A-1. PowerPC Instructions, by Execution Unit (Page 2 of 10)

Mnemonic	Description	Execution Unit[1]	Latency (cycles)[2]	Throughput (IPC)	Notes[3]	Comments
lswx (naturally aligned)	Load string word indexed	LSU	See *Table A-4* on page 726	1 per latency cycle	MC	Microcode assumes natural alignment and generates inline sequence of basic loads
lswi lswx stswi stswx (unaligned)	Load or store string word immediate Load or store string word indexed	LSU	See *Table A-4* on page 726	1 per latency cycle	MC	A string instruction is first decoded and issued as described above in the Programming Note. At execute, the LSU notes that it is unaligned and causes a machine flush. As the string instruction goes through microcode the second time, it is broken up in a way that takes the misalignment into account.
lwarx ldarx	Load and reserve indexed	LSU	See Note 2	See Note 2		Forced to miss data L1 cache. One outstanding **lwarx** in system at a time.
sync	Synchronize	LSU	See Note 2	See Note 2		The IU holds at issue until all queues and pipelines have drained for that thread. After issue, the **sync** forces previous stores to finish into the cache or memory hierarchy; that is, out of the STQs.
lwsync	Load synchronize	LSU	See Note 2	See Note 2		The IU holds at issue until all queues and pipelines have drained for that thread. After issue, the **lwsync** forces previous stores to finish into the cache or memory hierarchy; that is, out of the store queues. Still broadcasts **sync** transaction onto BE Bus (EIB) (but does not block)
eieio	Enforce in-order execution of I/O	LSU	See Note 2	See Note 2		
isync	Instruction synchronize	LSU	See Note 2	See Note 2	CSI	The IU holds at issue until all queues and pipelines have drained for that thread. Issued to the IU, and the IU performs a flush (N+1) when complete.
icbi	Instruction-cache block invalidate	LSU	See Note 2	See Note 2		After the LSU generates and translates the EA, the **icbi** looks like a snooped **icbi** to the instruction fetcher.
dcbt dbtst	Data-cache block touch	LSU	See Note 2	1 per latency cycle		

Table A-1. PowerPC Instructions, by Execution Unit (Page 3 of 10)

Mnemonic	Description	Execution Unit[1]	Latency (cycles)[2]	Throughput (IPC)	Notes[3]	Comments
dcbz	Instruction-cache block zero	LSU	See Note 2	1 per latency cycle		Invalidates the L1 cache line on its way to the L2. Allocation and zero function occur at the L2 cache.
dcbst	Data-cache block store	LSU	See Note 2	1 per latency cycle		
dcbf	Data-cache block flush	LSU	See Note 2	1 per latency cycle		
slbie	SLB invalidate entry	LSU	See Note 2	See Note 2		Causes class-based and thread-based invalidate in both the I-ERAT and the D-ERAT
slbia	SLB invalidate all	LSU	See Note 2	See Note 2		Fully invalidates the SLB, the I-ERAT, and D-ERAT
tlbie	TLB invalidate entry	LSU	See Note 2	See Note 2		Causes index-based invalidate in both the I-ERAT and the D-ERAT Broadcast onto the EIB.
tlbiel	TLB invalidate entry local	LSU	See Note 2	See Note 2		Causes index-based invalidate in both the I-ERAT and the D-ERAT Is not broadcast onto the EIB.
tlbsync	TLB synchronize	LSU	See Note 2	See Note 2		
slbmte	SLB move to entry	LSU	See *Table A-3* on page 724	See Note 2	MC	
slbmfev slbmfee	SLB move from entry VSID SLB move from entry ESID	LSU	See *Table A-3* on page 724	See Note 2	MC	
lha lhax lwa lwax	Load algebraic	LSU, FXU	See *Table A-4* on page 726	1 per latency cycle	MC	
lbzu lhzu lwzu ldu	Load with update	LSU, FXU	2 (RT) 2 (RA)	1 per latency cycle		Hardware breaks into basic **load** and an **add**.
lbzux lhzux lwzux ldux	Load and zero with update indexed	LSU, FXU	2 (RT) 2 (RA)	1 per latency cycle		Hardware breaks into a basic **load** and an **add**.
lhau	Load algebraic with update	LSU, FXU	See *Table A-4* on page 726	1 per latency cycle	MC	
lhaux lxaux	Load algebraic with update indexed	LSU, FXU	See *Table A-4* on page 726	1 per latency cycle	MC	

Table A-1. PowerPC Instructions, by Execution Unit (Page 4 of 10)

Mnemonic	Description	Execution Unit[1]	Latency (cycles)[2]	Throughput (IPC)	Notes[3]	Comments
stb sth stw std	Store	LSU, FXU	See Note 2	1 per latency cycle		
stbx sthx stwx stdx	Store indexed	LSU, FXU	See Note 2	1 per latency cycle		
stbu sthu stwu stdu	Store with update	LSU, FXU	2 for updated register	1 per latency cycle		Hardware breaks into basic **store** and an **add**.
sthbrx stwbrx	Store byte-reversed indexed	LSU, FXU	See Note 2	1 per latency cycle		
stbux sthux stwux stdux	Store with update indexed	LSU, FXU	2 for updated register	1 per latency cycle		Hardware breaks into basic **store** and an **add**.
stmw	Store multiple word	LSU, FXU	See *Table A-4* on page 726	1 per latency cycle	MC	Microcode generates inline sequence of basic stores.
stswi (naturally aligned)	Store string word immediate	LSU, FXU	See *Table A-4* on page 726	1 per latency cycle	MC	Microcode assumes natural alignment and generates inline sequence of basic stores.
stswx (naturally aligned)	Store string word indexed	LSU, FXU	See *Table A-4* on page 726	1 per latency cycle	MC	Microcode assumes natural alignment and generates inline sequence of basic stores.
stwcx. stdcx.	Store conditional indexed	LSU, FXU	See Note 2	See Note 2		Must establish coherency block ownership before completing the instruction (other stores do not have to do this). Can take anywhere from 10 cycles to hundreds of cycles depending upon the state of the coherency block in the memory hierarchy)
lfsu lfsux lfdu lfdux	Load floating-point with update	LSU, FXU, FPU Load	1 (fpr) 2 (gpr)	1 per latency cycle		
stfiwx	Store floating-point as integer indexed	LSU, FPU	See Note 2	1 per latency cycle		
lfs lfsx lfd lfdx	Load floating-point	LSU, FPU Load	1	1 per latency cycle		Instructions are issued to both the FPU and the LSU units; only executes in the LSU unit
stfs stfsx stfd stfdx	Store floating-point	LSU, FPU Store	See Note 2	1 per latency cycle		Instructions are dispatched to both the FPU and the LSU units
stfsu stfsux stfdu stfdux	Store floating-point with update	LSU, FPU, FXU	2 (gpr)	1 per latency cycle		

Table A-1. PowerPC Instructions, by Execution Unit (Page 5 of 10)

Mnemonic	Description	Execution Unit[1]	Latency (cycles)[2]	Throughput (IPC)	Notes[3]	Comments
addi addis add subf addco subfco addeo subfeo addmeo subfmeo addzeo subfzeo neg nego	Add Subtract Negate	FXU	2 (grp)	1 per latency cycle		
add. subf. subfic addze. neg. subfe	Add Subtract Negate	FXU	2 (gpr) 1 (cr)	1 per latency cycle		
addic. adde. subfe. addme. subfme. addze subfze.	Add carrying Add extended Subtract extended	FXU	2 (gpr) 1 (cr) 2 (xer)	1 per latency cycle		
addo. subfo. subfeo.	Add Subtract	FXU	2 (gpr) 1 (cr) 2 (xer)	1 per latency cycle		
addeo. addmeo. subfmeo. addzeo. subfzeo.	Add extended Subtract extended	FXU	2 (gpr) 1 (cr) 2 (xer)	1 per latency cycle		
addo subfo addc subfc addic subic adde addme subfme subfze	Add Subtract	FXU	2 (grp) 2 (xer)	1 per latency cycle		
addc. addco. subfc. subfco.	Add carry recording Subtract carry recording	FXU	2 (gpr) 1 (cr) 2 (xer)	1 per latency cycle		
mulli	Multiply immediate	FXU	6	1 per 6 latency cycles		Not pipelined in the FXU. Causes a 6-cycle stall after issue.
mullw mulhw mulhwu mullwo	Multiply word	FXU	9	1 per 9 latency cycles		Not pipelined in the FXU. Causes a 9-cycle stall after issue.
mulld mulhd mulhdu mulldo	Multiply doubleword	FXU	15	1 per 15 latency cycles		Not pipelined in the FXU. Causes a 15-cycle stall after issue.
mullw. mulld. mulhd. mulhw. mulhdu. mulhwu.	Multiply recording	FXU	See *Table A-3* on page 724	1 per latency cycle	MC	Not pipelined in the FXU. Microcode breaks into baseline operation and a compare.
mullwo. mulldo.	Multiply recording	FXU	See *Table A-3* on page 724	1 per latency cycle	MC	Not pipelined in the FXU. Microcode breaks into baseline operation and a compare.

Table A-1. PowerPC Instructions, by Execution Unit (Page 6 of 10)

Mnemonic	Description	Execution Unit[1]	Latency (cycles)[2]	Throughput (IPC)	Notes[3]	Comments

divd divdu divdo divduo	Divide	FXU (one)	10 - 70	1 per 10 to 1 per 70 latency cycles		Not pipelined in the FXU.
						The XU uses complex queue and pipeline stalls until write back.
						The fixed-point divide is a variable latency operation that calculates RA and RB for word or doubleword and signed or unsigned fixed-point (integer) operands. Division is defined by the following equation:
						`dividend = (quotient x divisor) + r`
						where:
						$0 \le r < \lvert divisor \rvert$, when dividend ≥ 0 and
						$-\lvert divisor \rvert < r \le 0$, when dividend < 0
						Overflow is set when an attempt is made to compute either the least negative integer divided by negative one or any integer divided by zero.
						The performance is determined by the number of bits required to represent the result.
						PPU cycles equal:
						`((1 setup) + (ceil ((rb leading digits - ra leading digits)/2) + 1 iterations) + (1 fixup)) × 2`
						word minimum = 10, maximum = 38 cycles
						doubleword minimum = 10, maximum = 70 cycles
						Overflow cases will complete in 10 cycles

divw divwu divwo divwuo	Divide	FXU (one)	10 - 38	1 per 10, to 1 per 38 latency cycles		Not pipelined in the FXU. The XU uses complex queue and pipeline stalls until write back. See the performance notes for **divd**.
divd. divw. divdu. divwu.	Divide recording	FXU (one)	See *Table A-3* on page 724	1 per latency cycle	MC	Not pipelined in the FXU. Microcode breaks into a divide and a compare. The XU uses complex queue and pipeline stalls until write back. See the performance notes for **divd**.

Table A-1. PowerPC Instructions, by Execution Unit (Page 7 of 10)

Mnemonic	Description	Execution Unit[1]	Latency (cycles)[2]	Throughput (IPC)	Notes[3]	Comments
divdo. divwo. divduo. divwuo.	Divide recording	FXU (one)	See *Table A-3* on page 724	1 per latency cycle	MC	Not pipelined in the FXU. Microcode breaks into a divide and a compare. See the performance notes for **divd**.
cmpi cmp cmpli cmpl cmpla	Compare	FXU	1	1 per latency cycle		
tdi twi td tw	Trap	FXU	See Note 2	1 per latency cycle		
ori oris xori xoris and or xor nand nor eqv andc orc	Logical	FXU	2 (gpr)	1 per latency cycle		
andi. andis. and. or. xor. nand. nor. eqv. andc. orc. nego.	Logical recording	FXU	See *Table A-3* on page 724	1 per latency cycle	MC	
extsb extsh extsw	Extend sign	FXU	2	1 per latency cycle		
extsb. extsh. extsw.	Extend sign recording	FXU	See *Table A-3* on page 724	1 per latency cycle	MC	
cntlzd cntlzw popcntb	Count leading zeros Population count	FXU	2 (gpr)	1 per latency cycle		
cntlzd. cntlzw.	Count leading zeros recording	FXU	See *Table A-3* on page 724	1 per latency cycle	MC	
rldicl rldic rldicr rlwinm rldimi rlwimi	Rotate left immediate	FXU	2 (gpr)	1 per latency cycle		
rldicl. rldicr. rldic. rlwinm. rldcl rldcl. rldcr rldcr. rlwnm rlwnm. rldimi. rldic. rlwimi.	Rotate left recording Rotate left clear	FXU	See *Table A-3* on page 724	1 per latency cycle	MC	
sld sld. slw slw. srd srd. srw srw. srad srad. sraw sraw. sradi. srawi.	Shift left Shift right	FXU	See *Table A-3* on page 724	1 per latency cycle	MC	
sradi srawi	Shift right algebraic immediate	FXU	2 (gpr)	1 per latency cycle		
mtspr(xer)	Move to XER	FXU (one)	See Note 2	See Note 2		

Table A-1. PowerPC Instructions, by Execution Unit (Page 8 of 10)

Mnemonic	Description	Execution Unit[1]	Latency (cycles)[2]	Throughput (IPC)	Notes[3]	Comments
mtcrf	Move to CR fields	FXU	See *Table A-3* on page 724	1 per latency cycle	MC	Bit 11 = 0
mtcrf		FXU	1	1 per latency cycle		Bit 11 = 1
mftb	Move from time base	FXU (one)	~40	~2 to 40 latency cycles		
mfspr (others) **mfmsr**	Move from SPR Move from MSR	either FXU or branch (depends on SPR)	varies based on SPR	varies based on SPR		LR and CTR have 1-cycle latency and can issue 1/cycle. Other SPRs are handled by the FXU and are nonpipelined. The pipeline stalls until writeback.
mtspr (others) **mtmsr mtmsrd**	Move to SPR Move to MSR	either FXU or BRU (depends on SPR)	varies based on SPR	varies based on SPR		**mtmsrd** L = '0' is microcoded. **mtmsrd** L = '1' is not. LR and CTR have 1-cycle latency and issue 1/cycle.
fmr fneg fabs fnabs fadd fadds fsub fsubs fmul fmuls	Floating-point move Floating-point absolute Floating-point negative Floating-point negative absolute Floating-point add Floating-point subtract Floating-point multiply	FPU	10	1 per latency cycle		
fmr. fneg. fabs. fnabs. fadd. fadds. fsub. fsubs. fmul. fmuls.	Floating-point move recording Floating-point absolute recording Floating-point negative recording Floating-point negative absolute recording Floating-point add recording Floating-point subtract recording Floating-point multiply recording	FPU	10 (fpr) +1 (cr)	1 per latency cycle		The IU creates stall conditions for CR1 dependencies.
fmadd fmadds fmsub fmsubs fnmadd fnmadds fnmsub fnmsubs	Floating-point multiply-add Floating-point multiply-subtract	FPU	10	1 per latency cycle		

Table A-1. PowerPC Instructions, by Execution Unit (Page 9 of 10)

Mnemonic	Description	Execution Unit[1]	Latency (cycles)[2]	Throughput (IPC)	Notes[3]	Comments
fmadd. fmadds. fmsub. fmsubs. fnmadd. fnmadds. fnmsub. fnmsubs.	Floating-point multiply-add recording Floating-point multiply-subtract recording	FPU	10 (fpr) +1 (cr)	1 cycle		The IU creates stall conditions for CR1 dependencies.
fdiv fdivs	Floating-point divide (IEEE)	FPU	74	1 per 74 latency cycles		
fdiv fdivs	Floating-point divide (non-IEEE)	FPU	56	1 per 56 latency cycles		Non-IEEE results are accurate to within 1 or 2 ULPs of the correctly rounded IEEE result, depending on rounding mode: 1 ULP for to-nearest and to-zero rounding modes (the most common), 2 ULP for the other modes.
fsqrt fsqrts	Floating-point square root (IEEE)	FPU	84	1 per 84 latency cycles		
fsqrt fsqrts	Floating-point square root (non-IEEE)	FPU	66	1 per 66 latency cycles		Non-IEEE results are accurate to within 1 or 2 ULPs of the correctly rounded IEEE result, depending on rounding mode: 1 ULP for to-nearest and to-zero rounding modes (the most common), 2 ULP for the other modes.
fres	Floating-point reciprocal estimate	FPU	10	1 per latency cycle		
frsqrte	Floating-point reciprocal square-root estimate	FPU	10	1 per latency cycle		Nonpipelined in the FPU
fdiv. fdivs. fsqrt. fsqrts. fres. frsqrte.	Floating-point divide recording Floating-point square root recording Floating-point reciprocal estimate recording	FPU	same as above +1 (cr)	1 per latency cycle		The IU creates stall condition for CR1 dependencies. Nonpipelined in the FPU
frsp	Floating-point round to single-precision	FPU	10	1 per latency cycle		
fctid fctidz fctiw fctiwz fcfid	Floating-point convert to integer Floating-point convert from integer	FPU	10	1 per latency cycle		
frsp. fctid. fctidz. fctiw. fctiwz. fcfid.	Floating-point convert to integer recording Floating-point convert from integer recording	FPU	same as above +1 (cr)	1 per latency cycle		The IU creates stall condition for CR1 dependencies.

Table A-1. PowerPC Instructions, by Execution Unit (Page 10 of 10)

Mnemonic	Description	Execution Unit[1]	Latency (cycles)[2]	Throughput (IPC)	Notes[3]	Comments
fcmpu fcmpo	Floating-point compare	FPU	1	1 per latency cycle		
fsel	Floating-point select	FPU	10	1 per latency cycle		
fsel.	Floating-point select recording	FPU	same as above +1 (cr)	1 per latency cycle		
mffs mffs.	Move from FPSCR	FPU	11-28	1 per 28 latency cycles		These instructions are stalled at VQ8 until all older VSU operations are complete.
mcrfs	Move to CR from FPSCR	FPU	14	1 per latency cycle		
mtfsfi mtfsfi. **mtfsf mtfsf.** **mtfsb0 mtfsb0.** **mtfsb1 mtfsb1.**	Move to FPSCR	FPU	1	1 per latency cycle		

1. The Execution Units are BRU: Branch unit, IU: Instruction unit, FPU: Floating-point unit, FXU: Fixed-point unit, LSU: Load and store unit.
2. Parameters with this note require elaborate descriptions and are best understood by taking a broader view of the machine as a whole. This may includes complex microcoded instructions or instructions that do not modify a register.
3. MC: Microcode, CSI: Context Synchronizing Instruction. A minimum of 11 cycles are required before the first instruction is received from the microcode ROM, so microcoded instructions should be avoided if possible. See *Section A.1.3* on page 723 for details about microcode.

Table A-2. Storage Alignment for PowerPC Instructions

Operand			Alignment		
Type	Size (bytes)	Byte Alignment	Within 8-Byte Block	Crosses 8-Byte Boundary	Crosses 32-Byte Boundary[1]
Integer load[5]	1, 2, 4, 8	any	optimal	optimal	poor (to microcode)
Integer store[5]	1, 2, 4, 8	any	optimal	optimal	poor (to microcode)
Floating-point load	4, 8	not word	poor (alignment interrupt)	poor (alignment interrupt)	poor (alignment interrupt)
Floating-point load	4, 8	word	optimal	optimal	poor (to microcode)
Floating-point store	4, 8	not word	poor (alignment interrupt)	poor (alignment interrupt)	poor (alignment interrupt)
Floating-point store	4, 8	word	optimal	optimal	poor (to microcode)
lmw, **lmd**, **stmw**, **stmd**	any multiple of 4 bytes (word) or 8 bytes (doubleword)	any	poor (to microcode)	poor (to microcode)	poor (to microcode)
load string word, store string word	any	any	poor (to microcode)	poor (to microcode)	poor (to microcode)
load string doubleword, store string doubleword	any	any	poor (to microcode)	poor (to microcode)	poor (to microcode)
Caching-inhibited load (not from microcode)	1, 2, 4, 8, 16[3]	natural alignment	optimal	optimal	N/A
Caching-inhibited load (not from microcode)	1, 2, 4, 8	not natural alignment	poor (alignment interrupt)	poor (alignment interrupt)	poor (alignment interrupt)
Caching-inhibited store (not from microcode)	1, 2, 4, 8, 16[3]	natural alignment	optimal	optimal	N/A
Caching-inhibited store (not from microcode)	1, 2, 4, 8	not natural alignment	poor (alignment interrupt)	poor (alignment interrupt)	poor (alignment interrupt)
Caching-inhibited load or store from microcode	1, 2, 4, 8, 16	any	poor (alignment interrupt)	poor (alignment interrupt)	poor (alignment interrupt)

1. Operations crossing 4-KB, 64-KB, or segment boundaries behave the same as crossing a 32-byte boundary.
2. For Load Vector Right and Store Vector Right that are quadword aligned, no attempt is made to access storage. On a load, the data returned is 0; on a store, the operation becomes a no-op.
3. Regular VXU operations are assumed to be aligned to the 16-byte boundary.
4. Due to the way these operations are defined, they do not cross a 32-byte boundary (data is accessed up to a16-byte boundary).
5. Byte-reversed loads and stores have no special alignment characteristics.

A.1.3 Microcoded Instructions

Instructions that are either too complex or require too many system resources to implement (such as load string instructions) are microcoded**.** This means that they are split into several simpler instructions (microwords).

Note: *A minimum of 11 cycles are required before the first instruction is received from the microcode ROM, so microcoded instructions should be avoided if possible.*

Most microcoded instructions are decoded into two or three simple PowerPC instructions, and they can be avoided in most cases. The microcoded instructions are typically decomposed into an integer and a load or store operation, with a dependency between them. Although most microcoded PowerPC instructions are decoded into only a few simple instructions, it is important to keep in mind that there are typically dependencies between the internal operations of the microcode, which generate stalls at the issue stage. Replacing the microcoded instructions with PowerPC instructions not only avoids stalling but also gives more latitude in scheduling instructions to avoid stalls, as well as potentially improving multithreaded performance.

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

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

A.1.3.1 *Unconditionally Microcoded instructions*

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

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

Table A-3 on page 724 describes the unconditionally microcoded instructions, except load and store instructions. *Table A-4* on page 726 describes the unconditionally microcoded load and store instructions. In both tables, the Latency column indicates the latency to access the first instructions. The Number Of Microwords column indicates the number of microwords generated, not the number of cycles required to execute the microwords, because the exact latency will depend on adjacent instructions.

The largest class of microcoded instructions are “CR recording” instructions, which set a value in the condition register (CR). These are composed of two instructions: the base operation followed by a dependent compare instruction. Because the compare is dependant on a general-purpose register (GPR) target produced by the arithmetic logical unit (ALU), the total amount of time taken is (`microcode-ROM access + 1 for the base instruction + 1 dependency stall + 1 compare instruction`). Some microcoded instructions, such as load and store string, and load and store multiple, are decomposed into many more operations.

Other things to note:

- If replacing a microcoded **or.** instruction with an **or** instruction followed by a **cmp**, be sure to avoid inadvertently changing the priority of a thread. See *Section 10 PPE Multithreading* on page 295 for details.
- The load and store string, and the load and store multiple, instructions can be replaced with a sequence of PowerPC load or store instructions, which can be an advantage if misses are expected in the data effective-to-real address translation (D-ERAT) buffer or the L1 data cache, and if interrupt latency is important.
- The load string instruction will first attempt to use load word instructions to move the data. If the access would cross a 32-byte boundary when it accesses the L1 data cache, the load will be flushed and refetched, and will proceed byte-by-byte. The store string instruction behaves similarly.

Table A-3. Unconditionally Microcoded Instructions (Except Loads and Stores) (Page 1 of 3)

Mnemonic	Class	Latency (cycles)	Number of Microwords	Comment
and.	CR recording	11	2	Record instructions are all handled the same way. The “root” instruction is issued followed by the cmpi_x instruction.
andc.	CR recording	11	2	
andi.	CR recording	11	2	The nonrecord form used in the microcode sequence is only available to microcode.
andis.	CR recording	11	2	The nonrecord form used in the microcode sequence is only available to microcode.
nand.	CR recording	11	2	
nor.	CR recording	11	2	
nego.	CR recording	11	2	
or.	CR recording	11	2	
orc.	CR recording	11	2	
xor.	CR recording	11	2	
cntlzd.	CR recording	11	2	
cntlzw.	CR recording	11	2	
divd.	CR recording	11	2	
divdo.	CR recording	11	2	
divdu.	CR recording	11	2	
divduo.	CR recording	11	2	

Table A-3. Unconditionally Microcoded Instructions (Except Loads and Stores) (Page 2 of 3)

Mnemonic	Class	Latency (cycles)	Number of Microwords	Comment
divw.	CR recording	11	2	
divwo.	CR recording	11	2	.
divwu.	CR recording	11	2	
divwuo.	CR recording	11	2	
eqv.	CR recording	11	2	
extsb.	CR recording	11	2	
extsh.	CR recording	11	2	
extsw.	CR recording	11	2	
mulhd.	CR recording	11	2	
mulhdu.	CR recording	11	2	
mulhw.	CR recording	11	2	
mulhwu.	CR recording	11	2	
mulld.	CR recording	11	2	
mulldo.	CR recording	11	2	
mullw.	CR recording	11	2	
mullwo.	CR recording	11	2	
nego.	CR recording	11	2	
rldcl	indirect rotate	11	4	All indirect shift and rotate instructions are handled using the same technique. First the mt_shr is issued, followed by two no-ops for delay, followed by the “root” instruction (that is, rldcl_sh).
rldcl.	CR recording	11	5	
rldcr	indirect rotate	11	4	
rldcr.	CR recording	11	5	
rldic.	CR recording	11	2	
rldicl.	CR recording	11	2	
rldicr.	CR recording	11	2	
rldimi.	CR recording	11	2	
rlwimi.	CR recording	11	2	
rlwinm.	CR recording	11	2	
rlwnm	indirect rotate	11	4	
rlwnm.	CR recording	11	5	
sld	indirect shift	11	4	
sld.	CR recording	11	5	
slw	indirect shift	11	4	
slw.	CR recording	11	5	
srad	indirect shift	11	4	

Table A-3. Unconditionally Microcoded Instructions (Except Loads and Stores) (Page 3 of 3)

Mnemonic	Class	Latency (cycles)	Number of Microwords	Comment
srad.	CR recording	11	5	
sradi.	CR recording	11	2	
sraw	indirect shift	11	4	
sraw.	CR recording	11	5	
srawi.	CR recording	11	2	
srd	indirect shift	11	4	
srd.	CR recording	11	5	
srw	indirect shift	11	4	
srw.	CR recording	11	5	
slbmte	miscellaneous	11	2	
slbmfev	miscellaneous	11	2	
slbmfee	miscellaneous	11	2	
mtcrf (bit 11 = ‘0’)	miscellaneous	11	9	
mcrfs	miscellaneous	11	2	
mtmsrd L = ’0’	miscellaneous	11	4	**mtmsrd** is microcoded if and only if the L field of the instruction is 0. The microcode routine is required for synchronization. The sequence is: **sync** L = ’1’; **mtmsrd** L = ’0’; **sync** L = ’1’; **isync;**
mtmsr L = ’0’	miscellaneous	11	4	**mtmsr** is microcoded if and only if the L field of the instruction is 0. The microcode routine is required for synchronization. The sequence is: **sync** L = ’1’; **mtmsr** L = ’0’; **sync** L = ’1’; **isync;**

Table A-4. Unconditionally Microcoded Loads and Stores (Page 1 of 2)

Mnemonic	Class	Latency (cycles)	Number of Microwords [1, 2]	Comment
lha	load algebraic	11	7	Handled by byte.
lhau	load algebraic	11	8	Handled by byte.
lhaux	load algebraic	11	8	Handled by byte.
lhax	load algebraic	11	8	Handled by byte.
lmw	load multiple	11	(2 + 1 × words)	This instruction is broken down into a series of load words.
lswi	load string/optimized	10	By word: (1 × words + 2 × bytes) By byte: (2 × bytes)	Optimized instruction [3]

Table A-4. Unconditionally Microcoded Loads and Stores (Page 2 of 2)

Mnemonic	Class	Latency (cycles)	Number of Microwords [1, 2]	Comment
lswx	load string/optimized	By word: 11 By byte: 7	By word: 4 + (1 × words + 2 × bytes) By byte: 4 + (2 × bytes)	Optimized instruction [3]
lwa	load algebraic	11	13	Handled by byte.
lwaux	load algebraic	11	12	Handled by byte.
lwax	load algebraic	11	12	Handled by byte.
stmw	store multiple	11	(2 + 1 × words)	Broken into a series of store words.
stswi	store string/optimized	10	By word: (1 × words + 2 × bytes) By byte: (2 × bytes)	Optimized instruction [3]
stswx	store string/optimized	7	By word: 4 + (1 × words + 2 × bytes) By byte: 4 + (2 × bytes)	Optimized instruction [3]

1. A microcode load or store operation can access an 8-bit byte or a 32-bit word, indicated as “by byte” or “by word” respectively.
2. “Words” means number of words.
3. The instruction is first broken down into a series of load-word instructions (odd bytes are handled by byte). If this does not cause an alignment exception, then the instruction is complete. If an alignment exception occurs, the first attempt is flushed. When the instruction is returned to microcode it is then handled a byte at a time. Odd bytes, if any, are defined as the remainder of string_count / 4. For store instructions, it is a series of store words.

A.1.3.2 *Conditionally Microcoded Instructions*

The L1 data cache is physically implemented with 32-byte sectors. Thus, a conditionally microcoded load or store instruction that attempts to perform an L1 data-cache access that crosses a 32-byte boundary must be split into several instructions. When one of these misaligned loads or stores first attempts to access the L1 data cache, the misalignment is detected and the pipeline (*Section A.6* on page 752) is flushed when the instruction reaches the EX7 stage. The flushed load or store is then refetched, converted to microcode at the decode stage, and split into the appropriate loads or stores, as well as any instructions needed to merge the values together into a single register.

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

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

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

All of the conditionally microcoded instructions are loads and stores. The details of these are shown in *Table A-5* on page 728.

Table A-5. Conditionally Microcoded Instructions (Page 1 of 2)

Mnemonic	Class	Latency (cycles)	Number of Microwords [1]	Comment
ld	load optimized	11	By word: 6 By byte: 18	Optimized instruction [2]
ldbrx	load	11	19	Optimized instruction [2]
ldu	load optimized	11	By word: 9 By byte: 21	Optimized instruction [2]
ldux	load optimized	11	By word: 7 By byte: 19	Optimized instruction [2]
ldx	load optimized	11	By word: 7 By byte: 19	Optimized instruction [2]
lfd	load	11	5	Handled by word or handled by byte
lfdu	load	11	6	Handled by word or handled by byte
lfdux	load	11	6	Handled by word or handled by byte
lfdx	load	11	6	Handled by word or handled by byte
lhbrx	load	11	7	Handled by byte.
lhz	load	11	6	Handled by byte
lhzu	load	11	7	Handled by byte
lhzux	load	11	7	Handled by byte
lhzx	load	11	7	Handled by byte
lwbrx	load	11	11	Handled by byte
lwz	load	11	10	Handled by byte
lwzu	load	11	11	Handled by byte
lwzux	load	11	11	Handled by byte
lwzx	load	11	11	Handled by byte
std	store optimized	11	By word: 6 By byte: 18	Optimized instruction [2]
stdbrx	store	11	18	New instruction
stfdux	store	11	5	Handled by word or handled by byte
stdu	store optimized	11	By word: 9 By byte: 21	Optimized instruction [2]
stdux	store optimized	11	By word: 6	Optimized instruction [2]

Table A-5. Conditionally Microcoded Instructions (Page 2 of 2)

Mnemonic	Class	Latency (cycles)	Number of Microwords [1]	Comment
stdx	store optimized	11	By byte: 18	Optimized instruction [2]
stfd	store	11	3	Handled by word
stfdu	store	11	5	Handled by word
stfdx	store	11	5	Handled by word
sth	store	11	5	Handled by byte
sthbrx	store	11	6	Handled by byte
sthu	store	11	7	Handled by byte
sthux	store	11	7	Handled by byte
sthx	store	11	6	Handled by byte
stw	store	11	4	Handled by byte
stwbrx	store	11	10	Handled by byte
stwu	store	11	11	Handled by byte
stwux	store	11	10	Handled by byte

1. A microcode load or store operation can access an 8-bit byte or a 32-bit word, indicated as “by byte” or “by word” respectively.
2. The instruction is first broken down into two load halfword instructions. If this does not cause a misalignment, then the instruction is complete. If a misalignment occurs, the first attempt is flushed. When the instruction is returned to microcode, it is then handled a byte at a time. For store instructions it is a series of store words.

A.2 PowerPC Extensions in the PPE

This section describes the differences between the instruction-set architecture of the PPE and version 2.0.2 of the *PowerPC Architecture*, as defined in the *PowerPC Architecture, Books I, II, and III, version 2.02* and further described in the *PowerPC Microprocessor Family: The Programming Environments for 64- and 32-Bit Microprocessors*. These differences include:

- New PowerPC instructions added in the PPE.
- New meaning to existing PowerPC instructions in the PPE.
- Optional PowerPC Instructions Implemented by the PPE.
- PowerPC Instructions Not Implemented by the PPE.
- Endian support.

A.2.1 New PowerPC Instructions

The PPE implements the following new instructions, relative to version 2.02 of the *PowerPC Architecture*:

- **ldbrx**—Load Double Word Byte Reverse Indexed X-form.

- **sdbrx**—Store Double Word Byte Reverse Indexed X-form.

Details are given starting on the next page.

Load Double Word Byte Reverse Indexed X-form

ldbrx **rt,ra,rb**

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

/ ... denotes a reserved field.

```
if RA = 0 then    b ← 0
else              b ← (RA)
EA ←      b +tea (RB)
RT ←      MEM(EA+tea7, 1) || MEM(EA+tea6, 1) || MEM(EA+tea5, 1) || MEM(EA+tea4, 1) ||
          MEM(EA+tea3, 1) || MEM(EA+tea2, 1) || MEM(EA+tea1, 1) || MEM(EA, 1)
```

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

That is, the bytes in memory aligned location as:

```
ABCDEFGH
```

will be stored in the GPR as:

```
HGFEDCBA
```

Special Registers Altered:

- None

Store Double Word Byte Reverse Indexed X-form

sdbrx **rt,ra,rb**

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

/ ... denotes a reserved field.

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

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

That is, bytes in GPR as:

```
ABCDEFGH
```

will be stored in a memory aligned location as:

```
HGFEDCBA
```

Special Registers Altered:

- None

A.2.2 Implementation-Dependent Interpretation of PowerPC Instructions

A.2.2.1 *No-Op Forms of OR and ORI Instructions*

The PPE supports a subset of the program priorities defined in the *PowerPC Operating Environment Architecture, Book III*, Section 3.4.1. The PPE also adds specific forms of the no-op instruction for performance enhancements.

For details about the program priorities and the no-op forms of the **or** and **ori** instructions that support them, see *Section 10.6.2.2 nop Instructions that Change Thread Priority and Dispatch Policy* on page 316.

A.2.2.2 *Logical Partitioning (LPAR)*

The PPE supports 32 different Logical Partitions (LPARs), each of which is a virtual machine managed by the Hypervisor. The Logical Partitioning Control Register (LPCR) has a new bit field [52] named Mediated Interrupt (MER). The associated interrupt is described in *Section 9 PPE Interrupts* on page 237. Values in the Real Mode Limit Select (RMLS) field [34:37] have the meanings given in *Table A-6*. All reserved values are treated as 1 MB.

Table A-6. Summary of Real Mode Limit Select (RMLS) Values

RMLS(0:2)	Effective Address Limit
0000	256 GB
0001	16 GB
0010	1 GB
0011	64 MB
0100	256 MB
0101	16 MB
0110	8 MB
0111	128 MB
1000	Reserved
1001	4 MB
1010	2 MB
1011	1 MB
11--	Reserved

A.2.2.3 *TLB Invalidation*

Software must use the **tlbie** or **tlbiel** instructions to invalidate entries in the Translation Lookaside Buffer (TLB). For these instructions, the PPE supports bits [22:(63 – *p*)] of the RB source register, where *p* is the page size. This support results in a selective invalidation in the TLB, based on VPN[38:(79 – *p*)] and the page size. Thus, any entry in the TLB with matching VPN[38:(79 – *p*)] and page size will be invalidated. For small pages (*p* = 12), the Effective to Real

Address Translation (ERAT) is also invalidated as a result of **tlbie** or **tlbiel** based on VPN[63:67], which is equivalent to EA[47:51]. Thus, any entry in the ERAT table with matching EA[47:51] is invalidated.

The PPE adds new fields to the RB register of the **tlbiel** instruction that are not currently defined in the *PowerPC Architecture*. These include the Large Page Selector (LP) and Invalidation Selector (IS) bits.

Table A-7 gives details of the implementation of the IS field. Bit 1 of the IS field is ignored. Bit 0 of the IS field is provided in RB[52] of the **tlbiel** instruction.

Table A-7. Summary of Supported IS Values in TLBIEL

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

The modified instruction definition for **tlbiel** specific to the PPE is shown on the next page.

TLB Invalidate Entry Local X-form

tlbiel **rb,l**

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

/, //, ///, ... denotes a reserved field.

```
inval_sel ← (RB)52
if inval_sel = 0 then
        if L = 0 then pg_size ← 4 KB
        else if L = 1 then
            LP ← (RB)51
        if LP = 0 then
            pg_size = large page size 1 selected by HID6[LB]16:17
        else if LP = 1 then
            pg_size = large page size 2 selected by HID6[LB]18:19
        p ← log_base_2(pg_size)
        for each TLB entry
            if (entry_VPN38:79-p = (RB)22:63-p) & (entry_pg_size = pg_size) then
                TLB entry ← invalid
    else if inval_sel = 1 then
        if entry_pg_size = 4 KB then
            if (entry_VPN52:55 / entry_VPN60:63 || entry_VPN64:67) = (RB)44:51 then
                TLB entry ← invalid
        else if entry_pg_size = 64 KB then
            if (entry_VPN52:55 / entry_VPN56:59 || entry_VPN60:63) = (RB)44:51 then
                TLB entry ← invalid
        else if entry_pg_size = 1 MB then
            if entry_VPN52:59 = (RB)44:51 then
                TLB entry ← invalid
        else if entry_pg_size = 16 MB then
            if entry_VPN48:55 = (RB)44:51 then
                TLB entry ← invalid
```

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

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

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

Since the ERAT table does not support large pages, any **tlbie** or **tlbiel** instruction to a large page (L = 1) will cause a complete invalidation of all entries in the ERAT table. For a **tlbie** or **tlbiel** instruction to a small page (L = 0), any I-ERAT or D-ERAT entries that have VPN[63:67] match RB[47:51] of the instruction will be invalidated.

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

The **tlbie** instruction register transfer language (RTL) is similar except that IS = 0 is assumed, and the instruction is performed on all processors belonging to the same partition as the processor issuing the **tlbie** instruction.

Since a **tlbiel** instruction to a large page causes the ERAT to be flushed, software may issue a **tlbiel** to a large page that is not currently in use (unmapped in the page table) to cause an ERAT flush. This is useful for invalidating ERAT entries during a partition context switch. The IS field of **tlbiel** should be set to b'00' for this operation.

For more information about the TLB, see *Section 4 Virtual Storage Environment* on page 79.

A.2.3 Optional PowerPC Instructions Implemented

The following optional PowerPC user-mode instructions are implemented in the PPE:

A.2.3.1 *Book I Optional Instructions Implemented*

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

A.2.3.2 *Book II Optional Instructions Implemented*

- **tlbie**—TLB Invalidate Entry (large and small page).
- **tlbiel**—Processor local form of TLB Invalidate Entry (large and small page).
- **tlbsync**—TLB Synchronize.
- **slbie**—SLB Invalidate Entry.
- **slbia**—SLB Invalidate All.
- **mtmsr**—Move to Machine State Register (32-bit).

A.2.4 PowerPC Instructions Not Implemented

The following PowerPC instructions are not implemented in the PPE:

A.2.4.1 *Book I Unimplemented Instructions*

The following instructions are obsolete and thus not implemented:

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

An attempt to execute one of these instructions will result in an illegal instruction-type exception.

A.2.4.2 *Book II Unimplemented Instructions*

The optional External Control Facility is not implemented in the PPE. Instructions associated with this facility are likewise not implemented. These include:

- **eciwx**—External Control In Word indeX.
- **ecowx**—External Control Out Word indeX.

An attempt to execute one of these instructions will result in an illegal instruction-type exception.

A.2.4.3 *Book III Unimplemented Instructions*

The External Access Register (EAR) is not implemented in the PPE. Instructions associated with the EAR are likewise not implemented. These include:

- **eciwx**—External Control In Word indeX.
- **ecowx**—External Control Out Word indeX.
- **tlbia**—TLB Invalidate All.
- **mtsr**—Move to Segment Register.
- **mtsrin**—Move to Segment Register Indirect.
- **mfsr**—Move from Segment Register.
- **mfsrin**—Move from Segment Register Indirect.

An attempt to execute one of these instructions will result in an illegal instruction-type exception.

A.2.5 Endian Support

The PPE does not support little-endian mode; it only supports big-endian mode. Writing to the Little-Endian mode (LE) and Interrupt Little-Endian mode (ILE) bits in the Machine State Register has no effect.

A.3 Vector/SIMD Multimedia Extension Instructions

An overview of the Vector/SIMD Multimedia Extension instructions is given in *Section 2.5* on page 55. The sections that follow summarize key points of the instructions. For a complete description, see the *PowerPC Microprocessor Family: Vector/SIMD Multimedia Extension Technolgy Programming Environments Manual*.

A.3.1 Data Types

Section 2.5.2 on page 57 contains a list of the Vector/SIMD Multimedia Extension data types.

A.3.2 Instructions

Table A-8 summarizes the Vector/SIMD Multimedia Extension instructions, showing their pipeline, latency, and throughput. Some comments on the table:

- An instruction can be executed in one or more execution units, as shown in *Table A-8*.
- Latency is the issue-to-issue latency for a dependent instruction. For example, if an **add** issues in cycle 20 and the soonest a dependent **xor** could issue is cycle 22, the latency for **add** is listed as 2. There are multiple latencies for some instructions when the instruction writes more than one type of register. In general, latency will be one cycle larger than use penalty.
- Throughput refers to the maximum sustained rate at which the PPE can execute instructions of the type noted in the absence of any dependencies and assuming infinite caches. It is shown as instructions per cycle (IPC).
- Some fields are labelled “N/A” (not applicable) to indicate that a more elaborate description is required. This includes complex microcoded instructions and instructions that do not modify a register.
- In the Execution Units column LSU is the load and store unit, and VXU is the Vector/SIMD Multimedia Extension unit.

Table A-8. Vector/SIMD Multimedia Extension Instructions (Page 1 of 4)

Instruction	Description	Execution Unit[1]	Latency	Throughput (IPC)	Comments
lvebx lvehx lvewx lvlx lvlxl lvrx lvrxl lvsl lvsr lvxl	Load vector indexed	VXU load, LSU	2	1	
stvebx stvehx stvewx stvlx stvlxl stvrx stvrxl stvx stvxl	Store vector indexed	VXU store, LSU	N/A	1	
dst dstt dstst dss dssall	Nop				Not implemented (these instructions are no longer part of the architecture. They will be treated as no-ops).
vpkpx vpkshss vpkshus vpkswss vpkuwus vpkuhum vpkuhus vpkuwum vpkuwus	Vector pack	permute	4	1	
vupkhpx vupkhsb vupkhsh vupklpx vupklsb vupklsh	Vector unpack	permute	4	1	
vmrghb vmrghh vmrghw vmrglb vmrglh vmrglw	Vector merge	permute	4	1	
vspltb vsplth vspltw vspltisb vspltish vspltisw	Vector splat	permute	4	1	
vperm	Vector permute	permute	4	1	
vsldoi	Vector shift left double by octet immediate	permute	4	1	

Table A-8. Vector/SIMD Multimedia Extension Instructions (Page 2 of 4)

Instruction	Description	Execution Unit[1]	Latency	Throughput (IPC)	Comments
vslo vsro	Vector shift left by octet	permute	4	1	
vaddubm vadduhm vadduwm vaddubs vadduhs vadduws vaddcuw vsububm vsubuhm vsubuwm vsububs vsubuhs vsubuws vsubsbs vsubshs vsubsws vsubcuw vsubsws vaddsbs vaddshs vaddsws	Vector add Vector subtract	simple	4	1	
vavgub vavguh vavguw vagvsb vagvsh vavgsw	Vector average	simple	4	1	
vand vor vxor vandc vnor	Vector logical	simple	4	1	
vsel	Vector select	simple	4	1	
vrlb vrlh vrlw vslb vslh vslw vsl vsrb vsrh vsrw vsr vsrab vsrah vsraw	Vector rotate Vector shift	simple	4	1	
vcmpgtub vcmpgtsb vcmpgtuh vcmpgtsh vcmpgtuw vcmpgtsw vcmpgtfp	Vector compare	simple	4	1	
vcmpgtub. vcmpgtsb. vcmpgtuh. vcmpgtsh. vcmpgtuw. vcmpgtsw. vcmpgtfp.	Vector compare recording	simple	4	1	
vcmpequb vcmpequh vcmpequw vcmpeqfp	Vector compare equal to	simple	4	1	
vcmpequb. vcmpequh. vcmpequw. vcmpeqfp.	Vector compare equal to recording	simple	4	1	
vcmpbfp vcmpgefp	Vector compare bounds Vector greater than or equal to	simple	4	1	
vcmpbfp. vcmpgefp.	Vector compare bounds recording Vector greater than or equal to recording	simple	4	1	

Table A-8. Vector/SIMD Multimedia Extension Instructions (Page 3 of 4)

Instruction	Description	Execution Unit[1]	Latency	Throughput (IPC)	Comments
vmaxub vmaxuh vmaxuw vmaxsb vmaxsh vmaxsw vmaxfp	Vector maximum	simple	4	1	
vminub vminuh vminuw vminsb vminsh vminsw vminfp	Vector minimum	simple	4	1	
mtvscr	Move the VSCR	simple	N/A	N/A	Stalls during issue until all older instructions are complete. Then, after issuing from the vector scalar unit issue queue, all younger vector scalar unit instructions stall until complete.
mfvscr	Move from VSCR	simple	N/A	N/A	Stalls during issue until all older instructions are complete.
vaddfp vsubfp vmaddfp vnmsubfp	Vector add single-precision Vector subtract single-precision Vector multiply-add single-precision Vector negative multiply-subtract single-precision	float	12	1	
vrefp vrsqrtefp	Vector reciprocal estimate single-precision Vector reciprocal square root estimate single-precision	estimate	14	1	

Table A-8. Vector/SIMD Multimedia Extension Instructions (Page 4 of 4)

Instruction	Description	Execution Unit[1]	Latency	Throughput (IPC)	Comments
vlogefp vexptefp	Vector log base-2 estimate floating-point Vector 2 raised to the exponent estimate floating-point	float	12	1	
vrfin vrfiz vrfip vrfim **vcfpsxws** **vcfpuxws** **vcuxwfp vcsxwfp**	Vector round Vector convert	float	12	1	
vmuloub vmulouh **vmulosb vmulosh** **vmuleub vmuleuh** **vmulesb vmulesh** **vmhaddshs** **vmhraddshs** **vmladduhm** **vmsumubm** **vmsummbm** **vmsumuhm** **vmsumuhs** **vmsumshm** **vmsumshs** **vsum4ubs** **vsum4sbs** **vsum4shs** **vsum2sws** **vsumsws**	Vector multiply Vector multiply-add Vector multiply-sum	complex	9	1	

1. The Vector/SIMD Multimedia Extension and FPU instructions are categorized into two groups. The first group, VSU type 1, includes Vector/SIMD Multimedia Extension simple, Vector/SIMD Multimedia Extension complex, Vector/SIMD Multimedia Extension float, and FPU arithmetic instructions. The second group, VSU type 2, include Vector/SIMD Multimedia Extension load, Vector/SIMD Multimedia Extension store, Vector/SIMD Multimedia Extension permute, FPU load, and FPU store instructions

Table A-9. Storage Alignment for Vector/SIMD Multimedia Extension Instructions

Operand			Alignment		
Type	Size (bytes)	Byte Alignment	Within 8-Byte Block	Crosses 8-Byte Boundary	Crosses 32-Byte Boundary[1]
VXU load (except **lvlx[l]**, **lvrx[l]**)	16	any	optimal	optimal	N/A[3]
VXU store (except **stvlx[l]**, **stvrx[l]**)	16	any	optimal	optimal	N/A[3]
VXU load or store left (**lvlx[l]**, **stvlx[l]**)	any	any	optimal	optimal	N/A[4]
VXU load or store right (**lvrx[l]**, **stvrx[l]**)	any	not quadword aligned	optimal	optimal	N/A[4]
VXU load or store right (**lvrx[l]**, **stvrx[l]**)	any	quadword aligned[2]	optimal	optimal	N/A[4]

1. Operations crossing 4-KB, 64-KB, or segment boundaries behave the same as crossing a 32-byte boundary.
2. For Load Vector Right and Store Vector Right that are quadword aligned, no attempt is made to access storage. On a load, the data returned is 0; on a store, the operation becomes a no-op.
3. Regular VXU operations are assumed to be aligned to the 16-byte boundary.
4. Due to the way these operations are defined, they do not cross a 32-byte boundary (data is accessed up to a16-byte boundary).

A.3.3 Graphics Rounding Mode

The Vector/SIMD Multimedia Extension instructions have a *graphics rounding mode* (enabled by default) that allows programs written with Vector/SIMD Multimedia Extension instructions to produce floating-point results that are equivalent in precision to those written in the SPU instruction set. In this Vector/SIMD Multimedia Extension mode, as in the SPU environment, the default rounding mode is r*ound to zero*, denormals are treated as zero, and there are no infinities or NaNs.

When the `HID1[grap_mode]` bit is set to 1, the VXU operates in graphics rounding mode.

The following set of rules apply to graphics rounding mode:

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

- If the result is less than the new negative_overflow_boundary, then the result will be x‘FFFF_FFFF’.

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

A.3.3.1 *Math Instructions (Multiply-Add, Multiply-Subtract, Add, Subtract)*

Since there are no infinity or NaN inputs, there is not a NaN result for `infinity - infinity` or for infinity × 0. The result of infinity or NaN multiplied by zero is zero.

A.3.3.2 *Floating-Point Compares*

NaN and Infinity inputs are treated as large contiguous numbers, and they are correctly compared. Comparisons of +0 to -0 are the same as those for non-graphics-rounding mode.

A.3.3.3 *Reciprocal Estimate*

The result of a reciprocal estimate is the same as in non-Java mode, except that NaN and Infinity inputs are interpreted as large numbers and translated to correct mathematical estimates.

A.3.3.4 *Reciprocal Square-Root Estimate*

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

A.3.3.5 *Logarithm Base-2 Estimate*

The same results as non-Java mode are generated except that NaN and Infinity inputs are interpreted as large numbers and translated to correct mathematical estimates. In non-Java mode, the log of a negative number is a NaN result. For negative inputs in graphics rounding mode, the result is the log of the absolute value of the input, except when the input is +0 or -0. If the input is +0, the output is x‘7FFF_FFFF’. If the input is -0, the output is x‘FFFF_FFFF’.

A.3.3.6 *Power-of-2 Estimate*

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

A.3.3.7 *Scaled Conversion to integer*

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

A.3.3.8 *Scaled Conversion from integer*

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

A.3.3.9 *Round to Floating-Point Integer*

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

A.4 C/C++ Language Extensions (Intrinsics) for Vector/SIMD Multimedia Extensions

An overview of the C/C++ intrinsics for the Vector/SIMD Multimedia Extension instructions is given in *Section 2.6* on page 59. The sections that follow summarize key points of the intrinsics.

A.4.1 Vector Data Types

The Vector/SIMD Multimedia Extension programming model adds a set of fundamental data types, called *vector types*, as shown in *Table A-10*.

Note: *Since the token, vector, is a keyword in the Vector/SIMD Multimedia Extension data types, it is recommended that the term not be used elsewhere in the program as, for example, a variable name.*

Table A-10. Vector/SIMD Multimedia Extension Data Types

Vector Data Type	Meaning	Values[1]
vector unsigned char	Sixteen 8-bit unsigned values	0 ... 255
vector signed char	Sixteen 8-bit signed values	-128 ... 127
vector bool char	Sixteen 8-bit unsigned boolean	0 (false), 255 (true)
vector unsigned short	Eight 16-bit unsigned values	0 ... 65535
vector unsigned short int	Eight 16-bit unsigned values	0 ... 65535
vector signed short	Eight 16-bit signed values	-32768 ... 32767
vector signed short int	Eight 16-bit signed values	-32768 ... 32767
vector bool short	Eight 16-bit unsigned boolean	0 (false), 65535 (true)
vector bool short int	Eight 16-bit unsigned boolean	0 (false), 65535 (true)
vector unsigned int	Four 32-bit unsigned values	0 ... 2^{32} - 1
vector signed int	Four 32-bit signed values	-2^{31} ... 2^{31} - 1
vector bool int	Four 32-bit unsigned values	0 (false), 2^{31} - 1 (true)
vector float	Four 32-bit single precision	IEEE-754 values
vector pixel	Eight 16-bit unsigned values	1/5/5/5 pixel

1. The represented values are in decimal (base-10) notation.

Introducing fundamental vector data types permits a compiler to provide stronger type-checking and supports overloaded operations on vector types.

A.4.2 Vector Literals

There are two standard formats of constructing a vector literal. A compiler may support one of both formats. The first format, as specified by the *AltiVec Technology Programming Interface Manual*, is shown in *Table A-11*. This format consists of a parenthesized vector type followed by a parenthesized set of constant expressions. The AltiVec format is deprecated and will often only be supported for legacy software.

Table A-11. AltiVec Vector-Literal Format

Notation	Description
(vector unsigned char)(unsigned int)	A set of 16 unsigned 8-bit quantities that all have the value specified by the integer.
(vector unsigned char)(unsigned int, ..., unsigned int)	A set of 16 unsigned 8-bit quantities specified by the 16 integers.
(vector signed char)(signed int)	A set of 16 signed 8-bit quantities that all have the value specified by the integer.
(vector signed char)(signed int, ..., signed int)	A set of 16 signed 8-bit quantities specified by the 16 integers.
(vector unsigned short)(unsigned int)	A set of 8 unsigned 16-bit quantities that all have the value specified by the integer.
(vector unsigned short)(unsigned int, ..., unsigned int)	A set of 8 unsigned 16-bit quantities specified by the 16 integers.
(vector signed short)(signed int)	A set of 8 signed 16-bit quantities that all have the value specified by the integer.
(vector signed short)(signed int, ..., signed int)	A set of 8 signed 16-bit quantities specified by the 16 integers.
(vector unsigned int)(unsigned int)	A set of 4 unsigned 32-bit quantities that all have the value specified by the integer.
(vector unsigned int)(unsigned int, ..., unsigned int)	A set of 4 unsigned 32-bit quantities specified by the 16 integers.
(vector signed int)(signed int)	A set of 4 signed 32-bit quantities that all have the value specified by the integer.
(vector signed int)(signed int, ..., signed int)	A set of 4 signed 32-bit quantities specified by the 16 integers.
(vector float)(float)	A set of 4 32-bit floating-point quantities that all have the value specified by the float.
(vector float)(float, float, float, float)	A set of 4 32-bit floating-point quantities that all have the value specified by the 4 floats.

The second format for constructing a vector literal is written as a parenthesized vector type followed by a curly-braced set of constant expressions, as shown in *Table A-12*. The elements of the vector are initialized to the corresponding expression. Elements for which no expressions are specified default to 0. This format is the preferred syntax and should be used for all new application software.

Table A-12. Curly-Brace Vector-Literal Format

Notation	Description
(vector unsigned char)*{unsigned int}*	A set of 16 unsigned 8-bit quantities that all have the value specified by the integer.
(vector unsigned char)*{unsigned int, ..., unsigned int}*	A set of 16 unsigned 8-bit quantities specified by the 16 integers.
(vector signed char)*{signed int}*	A set of 16 signed 8-bit quantities that all have the value specified by the integer.
(vector signed char)*{signed int, ..., signed int}*	A set of 16 signed 8-bit quantities specified by the 16 integers.
(vector unsigned short)*{unsigned int}*	A set of 8 unsigned 16-bit quantities that all have the value specified by the integer.
(vector unsigned short)*{unsigned int, ..., unsigned int}*	A set of 8 unsigned 16-bit quantities specified by the 16 integers.
(vector signed short)*{signed int}*	A set of 8 signed 16-bit quantities that all have the value specified by the integer.
(vector signed short)*{signed int, ..., signed int}*	A set of 8 signed 16-bit quantities specified by the 16 integers.
(vector unsigned int)*{unsigned int}*	A set of 4 unsigned 32-bit quantities that all have the value specified by the integer.
(vector unsigned int)*{unsigned int, ..., unsigned int}*	A set of 4 unsigned 32-bit quantities specified by the 16 integers.
(vector signed int)*{signed int}*	A set of 4 signed 32-bit quantities that all have the value specified by the integer.
(vector signed int)*{signed int, ..., signed int}*	A set of 4 signed 32-bit quantities specified by the 16 integers.
(vector float)*{float}*	A set of 4 32-bit floating-point quantities that all have the value specified by the float.
(vector float)*{float, float, float, float}*	A set of 4 32-bit floating-point quantities that all have the value specified by the 4 floats.

A.4.3 Intrinsics

The intrinsics are shown in *Table A-13* on page 746.

Table A-13. Vector/SIMD Multimedia Extension Intrinsics (Page 1 of 5)

Intrinsic	Description
Arithmetic	
d = vec_abs(a)	Vector Absolute Value
d = vec_abss(a)	Vector Absolute Value Saturated
d = vec_add(a,b)	Vector Add
d = vec_addc(a,b)	Vector Add Carryout Unsigned Word
d = vec_adds(a,b)	Vector Add Saturated
d = vec_avg(a,b)	Vector Average
d = vec_madd(a,b,c)	Vector Multiply Add

Table A-13. Vector/SIMD Multimedia Extension Intrinsics (Page 2 of 5)

Intrinsic	Description
d = vec_madds(a,b,c)	Vector Multiply Add Saturated
d = vec_max(a,b)	Vector Maximum
d = vec_min(a,b)	Vector Minimum
d = vec_mladd(a,b,c)	Vector Multiply Low and Add Unsigned Half Word
d = vec_mradds(a,b,c)	Vector Multiply Round and Add Saturated
d = vec_msum(a,b,c)	Vector Multiply Sum
d = vec_msums(a,b,c)	Vector Multiply Sum Saturated
d = vec_mule(a,b)	Vector Multiply Even
d = vec_mulo(a,b)	Vector Multiply Odd
d = vec_nmsub(a,b,c)	Vector Negative Multiply Subtract
d = vec_sub(a,b)	Vector Subtract
d = vec_subc(a,b)	Vector Subtract Carryout
d = vec_subs(a,b)	Vector Subtract Saturated
d = vec_sum4s(a,b)	Vector Sum Across Partial (1/4) Saturated
d = vec_sum2s(a,b)	Vector Sum Across Partial (1/2) Saturated
d = vec_sums(a,b)	Vector Sum Saturated
Rounding And Conversion	
d = vec_ceil(a)	Vector Ceiling
d = vec_ctf(a,b)	Vector Convert from Fixed-Point Word
d = vec_cts(a,b)	Vector Convert to Signed Fixed-Point Word Saturated
d = vec_ctu(a,b)	Vector Convert to Unsigned Fixed-Point Word Saturated
d = vec_floor(a)	Vector Floor
d = vec_trunc(a)	Vector Truncate
Floating-Point Estimate	
d = vec_expte(a)	Vector Is 2 Raised to the Exponent Estimate Floating-Point
d = vec_loge(a)	Vector Log2 Estimate Floating-Point
d = vec_re(a)	Vector Reciprocal Estimate
d = vec_rsqrte(a)	Vector Reciprocal Square Root Estimate
Compare	
d = vec_cmpb(a,b)	Vector Compare Bounds Floating-Point
d = vec_cmpeq(a,b)	Vector Compare Equal
d = vec_cmpge(a,b)	Vector Compare Greater Than or Equal
d = vec_cmpgt(a,b)	Vector Compare Greater Than
d = vec_cmple(a,b)	Vector Compare Less Than or Equal
d = vec_cmplt(a,b)	Vector Compare Less Than
Logical	
d = vec_and(a,b)	Vector Logical AND

Table A-13. Vector/SIMD Multimedia Extension Intrinsics (Page 3 of 5)

Intrinsic	Description
d = vec_andc(a,b)	Vector Logical AND with Complement
d = vec_nor(a,b)	Vector Logical NOR
d = vec_or(a,b)	Vector Logical OR
d = vec_xor(a,b)	Vector Logical XOR
Rotate and Shift	
d = vec_rl(a,b)	Vector Rotate Left
d = vec_round(a)	Vector Round
d = vec_sl(a,b)	Vector Shift Left
d = vec_sld(a,b,c)	Vector Shift Left Double
d = vec_sll(a,b)	Vector Shift Left Long
d = vec_slo(a,b)	Vector Shift Left by Octet
d = vec_sr(a,b)	Vector Shift Right
d = vec_sra(a,b)	Vector Shift Right Algebraic
d = vec_srl(a,b)	Vector Shift Right Long
d = vec_sro(a,b)	Vector Shift Right by Octet
Load and Store	
d = vec_ld(a,b)	Vector Load Indexed
d = vec_lde(a,b)	Vector Load Element Indexed
d = vec_ldl(a,b)	Vector Load Indexed LRU
d = vec_lvsl(a,b)	Vector Load for Shift Left
d = vec_lvsr(a,b)	Vector Load Shift Right
vec_st(a,b,c)	Vector Store Indexed
vec_ste(a,b,c)	Vector Store Element Indexed
vec_stl(a,b,c)	Vector Store Indexed LRU
Pack and Unpack	
d = vec_pack(a,b)	Vector Pack
d = vec_packpx(a,b)	Vector Pack Pixel
d = vec_packs(a,b)	Vector Pack Saturated
d = vec_packsu(a,b)	Vector Pack Saturated Unsigned
d = vec_unpackh(a)	Vector Unpack High Element
d = vec_unpackl(a)	Vector Unpack Low Element
Merge	
d = vec_mergeh(a,b)	Vector Merge High
d = vec_mergel(a,b)	Vector Merge Low
Permute and Select	
d = vec_perm(a,b,c)	Vector Permute
d = vec_sel(a,b,c)	Vector Select

Table A-13. Vector/SIMD Multimedia Extension Intrinsics (Page 4 of 5)

Intrinsic	Description
Stream	
vec_dss(a)	Vector Data Stream Stop
vec_dssall()	Vector Stream Stop All
vec_dst(a,b,c)	Vector Data Stream Touch
vec_dstst(a,b,c)	Vector Data Stream Touch for Store
vec_dststt(a,b,c)	Vector Data Stream Touch for Store Transient
vec_dstt(a,b,c)	Vector Data Stream Touch Transient
Move	
d = vec_mfvscr	Vector Move from Vector Status and Control Register
vec_mtvscr(a)	Vector Move to Vector Status and Control Register
Replicate	
d = vec_splat(a,b)	Vector Splat
d = vec_splat_s8(a)	Vector Splat Signed Byte
d = vec_splat_s16(a)	Vector Splat Signed Half-Word
d = vec_splat_s32(a)	Vector Splat Signed Word
d = vec_splat_u8(a)	Vector Splat Unsigned Byte
d = vec_splat_u16(a)	Vector Splat Unsigned Half-Word
d = vec_splat_u32(a)	Vector Splat Unsigned Word
All Predicates	
d = vec_all_eq(a,b)	All Elements Equal
d = vec_all_ge(a,b)	All Elements Greater Than or Equal
d = vec_all_gt(a,b)	All Elements Greater Than
d = vec_all_in(a,b)	All Elements in Bounds
d = vec_all_le(a,b)	All Elements Less Than or Equal
d = vec_all_lt(a,b)	All Elements Less Than
d = vec_all_nan(a)	All Elements Not a Number
d = vec_all_ne(a,b)	All Elements Not Equal
d = vec_all_nge(a,b)	All Elements Not Greater Than or Equal
d = vec_all_ngt(a,b)	All Elements Not Greater Than
d = vec_all_nle(a,b)	All Elements Not Less Than or Equal
d = vec_all_nlt(a,b)	All Elements Not Less Than
d = vec_all_numeric(a)	All Elements Numeric
Any Predicates	
d = vec_any_eq(a,b)	Any Element Equal
d = vec_any_ge(a,b)	Any Element Greater Than or Equal
d = vec_any_gt(a,b)	Any Element Greater Than
d = vec_any_le(a,b)	Any Element Less Than or Equal

Table A-13. Vector/SIMD Multimedia Extension Intrinsics (Page 5 of 5)

Intrinsic	Description
d = vec_any_lt(a,b)	Any Element Less Than
d = vec_any_nan(a)	Any Element Not a Number
d = vec_any_ne(a,b)	Any Element Not Equal
d = vec_any_nge(a,b)	Any Element Not Greater Than or Equal
d = vec_any_ngt(a,b)	Any Element Not Greater Than
d = vec_any_nle(a,b)	Any Element Not Less Than or Equal
d = vec_any_nlt(a,b)	Any Element Not Less Than
d = vec_any_numeric(a)	Any Element Numeric
d = vec_any_out(a,b)	Any Element Out of Bounds

A.5 Issue Rules

The PPE supports dual-issue of PowerPC and Vector/SIMD Multimedia Extension instructions at both the pipeline stages at which instructions are issued. However, there are several restrictions that limit which instructions can be issued together.

Figure A-1 on page 751 illustrates valid issue combinations. The instruction in slot 0 (the older instruction) is shown on the left vertical axis and the instruction in slot 1 (the younger instruction) is shown on the top horizontal axis. The pink squares demonstrate valid dual-issue combinations, and the white squares indicate illegal combinations. The Vector/SIMD Multimedia Extension unit (VXU) and Floating-Point Unit (FPU) instructions are categorized into two groups:

- Type 1: VXU simple, VXU complex, VXU floating-point, and FPU arithmetic instructions.
- Type 2: VXU load, VXU store, VXU permute, FPU load, and FPU store instructions.

Also, some instructions are classified in more than one group. For example, all fixed-point store and load-with-update instructions require both the Fixed-Point Unit (FXU) and the Load and Store Unit (LSU) in order to execute. In this case, both rows or columns for FXU and LSU must be checked to see if an issue conflict will occur. The classification of each *PowerPC Architecture* instruction can be found in *Table A-1* on page 712.

Figure A-1. Dual-Issue Combinations

Slot 0 \ Slot 1	FXU	LSU	BRU	VSU Type 1	VSU Type 2
FXU					
LSU					
BRU					
VSU Type 1					
VSU Type 2					

BRU	Branch Unit
FPU	Floating-Point Unit
FXU	Fixed-Point Unit
LSU	Load and Store Unit
VSU	Vector Scalar Unit (a combination of VXU and FPU)
VXU	Vector/SIMD Multimedia Extension Unit

There are several specific cases that prevent dual instruction issue. These are:

- A dependency stall exists. Input parameters are not yet available.
- The even instruction (in slot 0—the older instruction in a dispatch pair) is a nonpipelined instruction. The nonpipelined instructions include **mfspr**, **mulli**, **mullw**, **mullwo**, **mulhu**, **mulhwu**, **mulld**, **mulldo**, **mulhd**, **mulhdu**, **divd**, **divdu**, **divdo**, **divduo**, **fdiv**, **fdivs**, **fsqrt**, **fsqrts**, and **mtvscr**.
- The even instruction is a context-synchronizing instruction. The context-synchronizing instructions include **sync** (L=0,1,or 2), **isync**, **mtctrl**, **mtmsr** and **mtmsrd** (L=0), sc, rfid, and **hrfid**.
- One of the instructions is a microcoded instruction. The PPE has two types of microcoded instructions—unconditional and conditional. The unconditional microcoded instructions include shifts and rotates that source the shift amount from a register (as apposed to an immediate field), load/store algebraic instructions, load/store strings and multiples, and several CR recording (Rc=1) instructions. The conditional microcoded instructions are all load and store instructions and occur conditionally based upon the alignment of the load or store.

- The FPU is running single-step as a result of precise floating-point exceptions being enabled (machine-state register bits `MSR[FE0]` or `MSR[FE1]` being set to 1).
- The issue queue for the Vector Scalar Unit (VSU, a combination of the VXU and FPU) is single-stepping as a result of an internal flush. Internal flushes occur when VSU instructions have denormalized operands or produce denormalized results.
- A VXU floating-point instruction (except for a reciprocal-estimate operation) is issued within two cycles following a reciprocal estimate instruction, **vrefp** or **vrsqrtefp**.
- A **mffs** (move from FPSCR) instruction is issued one cycle following an FPU instruction, other than an FPU load or store.
- No instruction can be issued following a nonpipelined FPU or VXU instruction until the nonpipelined instruction completes.
- No FPU instruction, including loads and stores, can be issued after a **fdiv** or **fsqrt** until the **fdiv** or **fsqrt** completes.
- A younger (slot 1) VSU load or store can be issued in the same cycle as any other VSU type 1 instruction, but an older VXU or FPU load or store cannot.
- Nothing issues while an internal flush condition is resolved. During this time, instructions are issued from the denorm-recycle queue in a single-step fashion.

Additional information can be found in *Section A.7* on page 757.

A.6 Pipeline Stages

The Instruction Unit (IU) is responsible for all instruction handling and control. After instructions are issued from the IU, they enter one of the execution units (LSU, FXU, VXU, FPU, or BRU). The BRU is an execution unit, but is considered part of the IU because it deals with instruction branching.

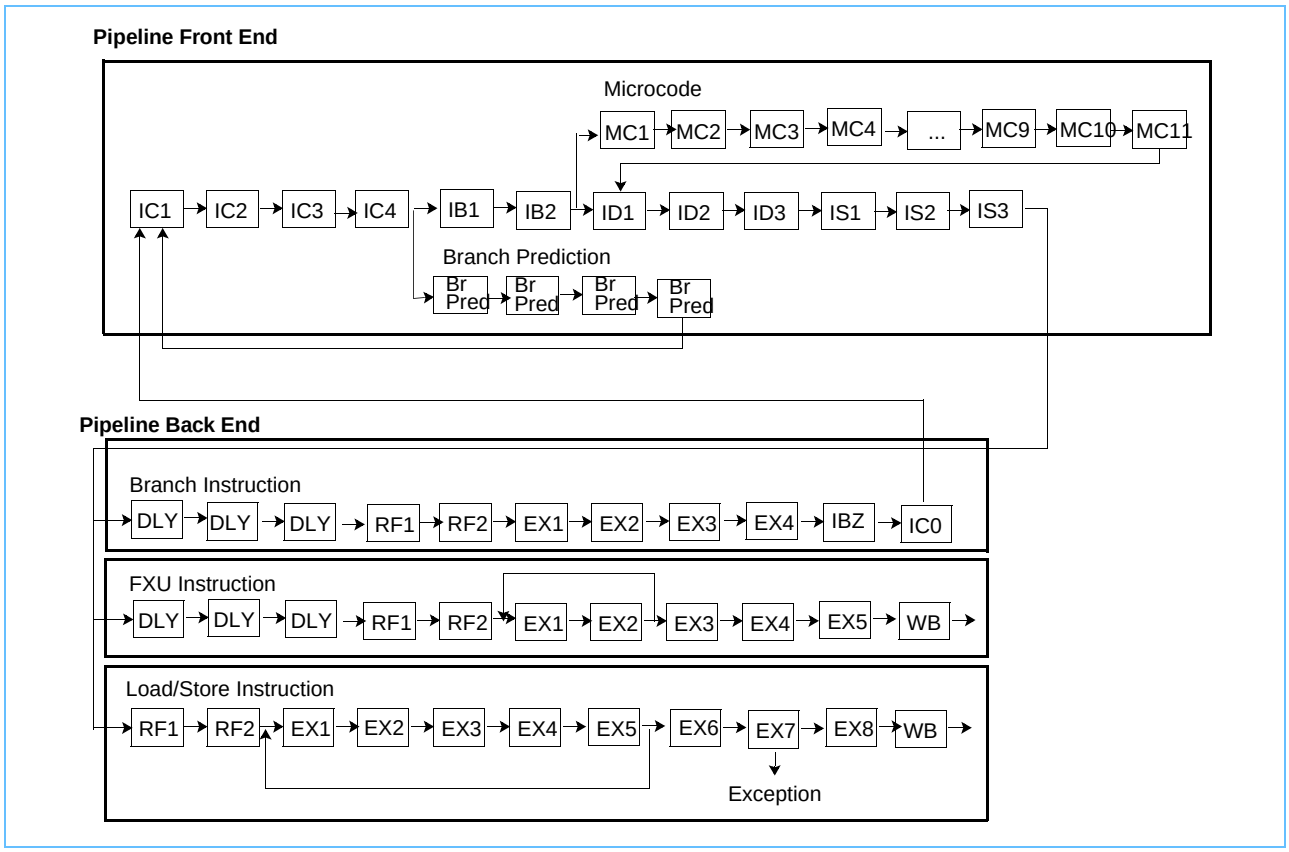
A.6.1 Instruction-Unit Pipeline

Figure A-2 on page 753 shows the portion of the IU pipeline that serves of the BRU, FXU, and LSU. The front end of the IU pipeline consists of a 4-cycle cache access: IC1 through IC4. The fetch address is based on either the sequential path or the branch target path. The branch execution stage can correct the instruction fetch with the branch-target address if the branch is mispredicted not taken. The branch execution stage can correct the instruction fetch with the sequential address if the branch is mispredicted taken.

After pipeline stage IC4, there are two instruction-buffer stages: IB1 and IB2. The instruction buffer (IBuf) is a FIFO queue. It is used to buffer the four instructions fetched from the L1 ICache when there is a downstream stall condition. Instruction-buffer stage IB1 is used to load the IBuf; there is one set of IBufs for each thread. IB2 is used to unload the IBuf and multiplex down to two instructions (this is the instruction dispatch stage). Each thread is given equal priority in dispatch, toggling every other cycle, unless specified otherwise in software (see *Section 10 PPE Multithreading* on page 295). If one thread is not fetching instructions into the IBuf (due to conditions such as a cache miss or thread disabled), then the other thread may have exclusive access to dispatch. Dispatch also controls the flow of instructions to and from microcode. Microcode (MC) is used to break an instruction that is difficult to execute into multiple smaller operations.

The stages labeled RF are GPR read cycles. Those labeled EX and execution cycles. Those labeled WB are write-back.

Figure A-2. BRU, FXU, and LSU Pipeline for PPE



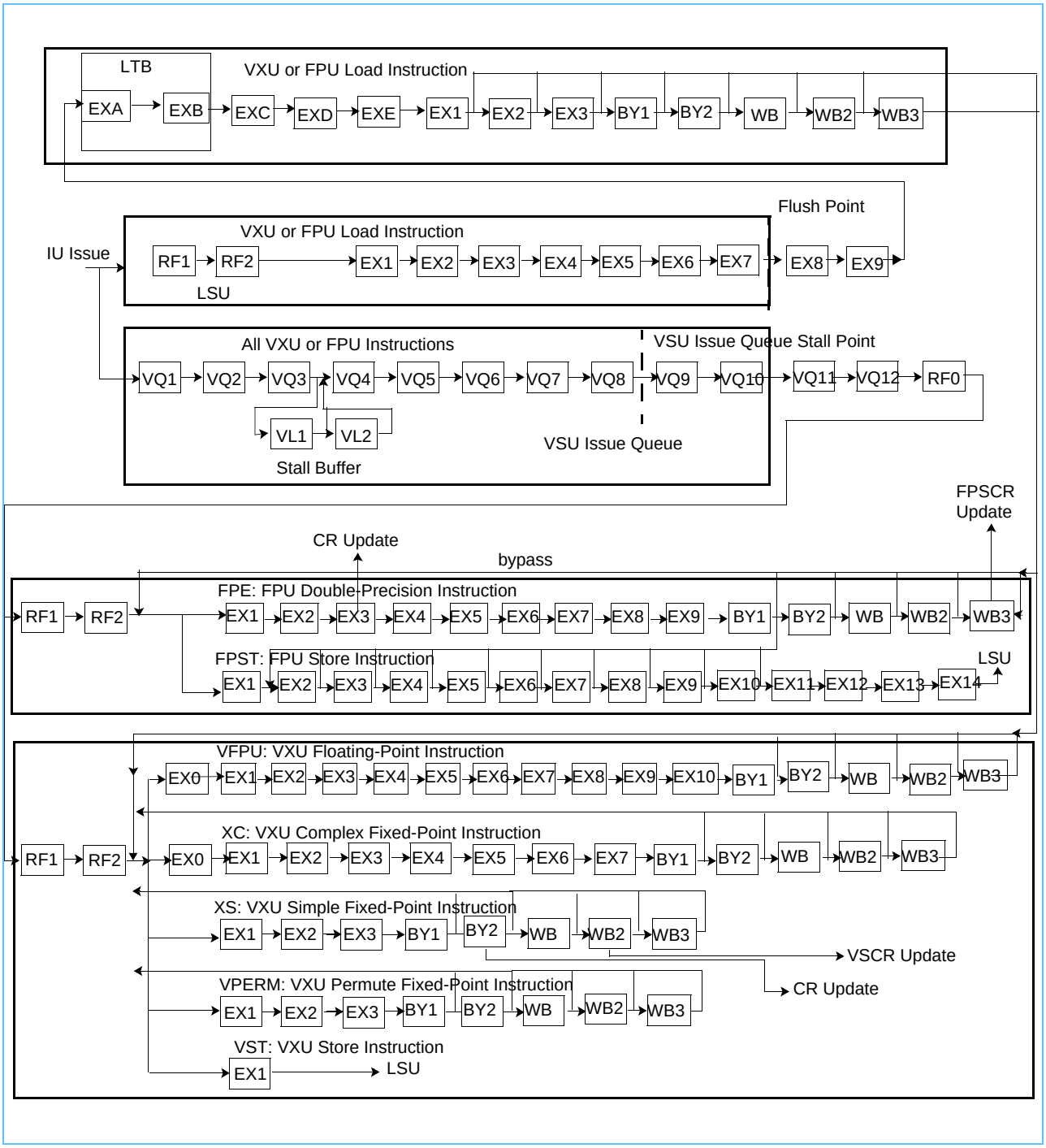
Dispatch grouping occurs based on the effective address bits [60:63] of the instruction. The dispatcher groups instructions with addresses x‘0’ and x‘4’, as well as instructions with addresses x‘8’ and x‘C’. An instruction with address x‘0’ cannot be grouped with an instruction with address x‘8’ or x‘C’.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

Instruction-decode pipeline stages ID1 through ID3 are used to assemble the internal opcodes and register source and target fields. In addition, dependency checking starts in ID2, which checks for data hazards, such as read-after-write (RAW) or write-after-write (WAW). The issue logic continues in ID3, IS1, and IS2 to create a single stall point at IS2, which is propagated up the pipeline to the IBufs, stalling both threads. The IS2 stall point is driven by data-hazard detection, in addition to resource-conflict detections, among other conditions. The IS2 issue stage determines the appropriate routing of the instructions. Then, they are issued to the execution units in the IS3 cycle. Each instruction in IS2 can be routed to five different issue slots: to the FXU, LSU, BRU, and to two slots in the VSU issue queue, described in the next section.

A.6.2 Vector/Scalar Unit Issue Queue

Figure A-3 shows the VSU Issue Queue (VIQ). VSU instructions are issued from the IU's IS2 stage to the VIQ without checking for dependencies or issue restrictions. The VIQ has a separate dependency-checking facility, used exclusively for VSU instructions. A separate stall point is generated at the bottom of the VIQ in cycle VQ8.

Figure A-3. VSU (FPU and VXU) Pipeline for PPE



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

After VQ8, instructions are sent to the VXU or FPU. The VQ9, VQ10, VQ11, and VQ12 stages are transmit cycles required to physically send the data to the execution unit.

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

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

If a VXU or FPU load misses the L1 DCache, it is issued to the VXU and FPU Load Miss Queue (VMQ). It is held in the VMQ until the cache miss is resolved in the LSU. When the miss is resolved, the load is reissued to the VSU.

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

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

A.6.3 Stall and Flush Points

In normal instruction execution, everything will flow in an orderly pipelined fashion. Several conditions exist in which this normal execution is interrupted with a stall or a flush condition. There are three separate stall points for the PPE.

The first stall point occurs just after the instruction buffers in the IB2 stage. This stall point is commonly referred to as a *dispatch block* because it prevents instructions from being dispatched. There are several possible reasons for a dispatch block. The following are two of the more common reasons:

- A special **nop** instruction (see *Section 10.6.2.2 nop Instructions that Change Thread Priority and Dispatch Policy* on page 316).

- The flush logic, whenever an L1 DCache miss dependency (RAW or WAW) or D-ERAT miss occurs; or whenever a Condition Register Field 0 (CR0) source-dependent operation issues while an **stdcx.** or **stwcx.** instruction is pending.

Stalling at this point in the pipeline allows the other thread to continue to dispatch instructions while the current thread is stalled. All other stall points in the pipeline block both threads.

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

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

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

The third stall point is exclusive to the VIQ (in cycle VQ8). It is activated for the following conditions:

- VXU or FPU dependencies.
- VXU or FPU write-port conflicts on the Vector Registers (VRs) or Floating-Point Registers (FPRs), invalid dual-issue combinations.
- Nonpipelined instruction issue.
- Special single-stepping conditions.

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

A 2-stage buffer separates the third stall point from the second stall point to help decouple the FXU and LSU instruction flow from the VXU and FPU instruction flow. If this buffer fills up and there is an instruction in the IS2 stage that needs to issue to the VIQ, then the second stall point will activate to prevent overflowing the VIQ.

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

The VXU and FPU contain an internal flush for instructions that have denormalized operands. (For more information about denormalization, see *PowerPC User Instruction Set Architecture, Book I.)* In this case, all instructions that have issued out of the VIQ and are younger than the denormalized instruction will be invalidated and then reexecuted. To facilitate this, there is a

Denormalized Instruction Queue (DIQ) that tracks all instructions that are issued from the VIQ. When the internal flush condition is signaled, the VIQ is stalled. Instructions in the DIQ are reissued one at a time (each instruction must complete before the next is issued).

A.7 Compiler Optimizations

The following compiler optimizations apply to all software written for the PPE, using either the PowerPC or Vector/SIMD Multimedia Extension instruction sets. These optimizations will improve the performance of the PPE.

For many additional compiler optimizations, including those specifically for SIMD operations, see *Section 22* on page 619.

A.7.1 Instruction Arrangement

PPE instructions are dispatched in 8-byte aligned pairs. The smaller address of the pair is termed I0 (EA[60:63] = x'0 or x'8); the larger address of the pair is termed I1 (EA[60:63] = x'4 or x'C). By convention, I0 is termed "older" than I1 and I1 is termed "younger" than I0.

See *Section A.5* on page 750 for an overview of issue rules and the names of execution units. The following rules apply:

- I0 and I1 should not attempt to make use of the same execution unit (BRU, LSU, FXU, VSU type 1, or VSU type 2). Doing so causes a 1-cycle stall for I1.
- I0 and I1 should not both access (read or write) the CR or LR registers. Doing so causes a 1-cycle stall for I1.
- If I0 is a load or store instruction and I1 is a any Vector/SIMD Multimedia Extension or FPU instruction other than a load or store and if I1 is not dependent on I0, then the order of I0 and I1 should be reversed. Failure to do so will cause a 1-cycle stall for I1.
- Nonpipelined operations (see below) and context-synchronizing instructions (CSI, see below) should be placed in I1. If placed in I0, then the I1 instruction can not dual-issue with the second-issue of the double-issue op (causing a 1 cycle stall penalty for the I1 instruction.
- The target of a branch instruction should be to an x'0 or x'8 (I0) address. This allows the dispatcher to avoid an unpaired instruction (resulting in a 1-cycle inefficiency).
- Predicted-taken branches should have an address of x'4 or x'8 for maximum dispatch efficiency.

A.7.2 Avoiding Slow Instructions and Processor Modes

For better performance, software should avoid setting MSR[FE0] or MSR[FE1] to '1'. When either of these are set, the processor will run in a "single-step" mode whereby each instruction must wait for all older instructions to complete before getting issued.

Because of the 11-cycle penalty for beginning a microcode sequence, it is best for performance if software avoids instructions that go to microcode. See *Table A-3* on page 724, *Table A-4* on page 726, and *Table A-5* on page 728.

The following are context-synchronizing instructions (CSI). These instructions require all older instructions to complete before they are issued, and are therefore best to avoid in performance-critical code:

- **sync** (L = 0, 1, or 2)
- **isync**
- **mtctrl**
- **mtmsr**, **mtmsrd** L = 0
- **sc**
- **rfid**, **hrfid**

The following are nonpipelined instructions. These instructions require all younger instructions to stall until they are completed:

- **mfspr** LR or **mfspr** CTR—inserts a 2-cycle stall.
- all other mfspr instructions—inserts a minimum 9-cycle stall.
- **mulli**—inserts a 6-cycle stall.
- **mullw**, **mullwo**, **mulhw**, **mulhwu**—inserts a 9-cycle stall.
- **mulld**, **mulldo**, **mulhd**, **mulhdu**—inserts a 15-cycle stall.
- **divd**, **divdu**, **divdo**, **divduo**—inserts a 10 to 70 cycle stall.
- **divw**, **divwu**, **divwo**, **divwuo**—inserts a 10 to 70 cycle stall.
- **fdiv**, **fdivs**, **fsqrt**, **fsqrts**, **mtvscr**—inserts a minimum 13 cycle stall.

The Vector/SIMD Multimedia Extension and FPU instructions that have either denormalized or underflow (NaN) operands or results should be avoided. If such an instruction is encountered, an internal flush will occur in the VIQ, resulting in a temporary single-step mode. This will cost a minimum of 100 cycles.

A.7.3 Avoiding Dependency Stalls and Flushes

The following dependencies should be avoided if possible:

- An instruction that reads the XER while a **stwcx.** or **stdcx.** instruction is pending will be flushed.
- An instruction that reads the XER should not be I1 in the same dispatch group or I0 or I1 in the following dispatch group as an XER writing instruction. If so, the issue logic will stall in order to separate them by a 1-cycle bubble.
- Two instructions that read or write the CTR should not be in the same or back-to-back dispatch groups. If so, the issue logic will stall in order to separate them by a 1-cycle bubble.
- In general, FXU instructions that read the CR should not access the same CR fields that are in use by Vector/SIMD Multimedia Extension or FPU instructions. Doing so can cause a flush if the FXU instruction issues before the Vector/SIMD Multimedia Extension or FPU instruction has written results to the CR field.
- An FXU instruction that is dependent on an older instruction should not be in the same or the following dispatch group as the older instruction. If so, the issue logic will stall in order to separate them by a 1-cycle bubble.

- An LSU instruction that is dependent on an older LSU instruction should not be in the same or the following dispatch group as the older instruction. If so, the issue logic will stall in order to separate them by a 1-cycle bubble.
- An LSU instruction that is dependent on an older FXU instruction should not be in the same or the following four dispatch groups as the older instruction. If so, the issue logic will stall in order to separate them by a 4-cycle bubble.

  There is an exception for a store instruction that has only data (not an address) dependent on an older FXU instruction. In this case the store should not be in the same or the following dispatch group. If it is, then a 4-cycle bubble will occur (even though it appears that only a 1-cycle bubble should be necessary).

For Vector/SIMD Multimedia Extension and FPU instructions, the following read-after-write (RAW) dependencies should be avoided if possible:

- An instruction dependent on an older FPU Arithmetic result should not be in the same or the following nine dispatch groups as the older instruction. If so, the issue logic will stall in order to separate them by a 9-cycle bubble. The exception is if the dependent instruction is an FPU Store which should not be in the same dispatch group as the FPU Arithmetic (or a 1 cycle stall will result).
- An instruction dependent on an older Vector/SIMD Multimedia Extension Simple or Vector/SIMD Multimedia Extension Permute result should not be in the same or the following three dispatch groups as the older instruction. If so, the issue logic will stall in order to separate them by a 3-cycle bubble.
- An instruction dependent on an older Vector/SIMD Multimedia Extension Complex result should not be in the same or the following eight dispatch groups as the older instruction. If so, the issue logic will stall in order to separate them by a 3-cycle bubble.
- An instruction dependent on an older Vector/SIMD Multimedia Extension Float result should not be in the same or the following eleven dispatch groups as the older instruction. If so, the issue logic will stall in order to separate them by a 11-cycle bubble.
- An instruction dependent on an older Vector/SIMD Multimedia Extension Estimate result should not be in the same or the following thirteen dispatch groups as the older instruction. If so, the issue logic will stall in order to separate them by a 13-cycle bubble.

For Vector/SIMD Multimedia Extension and FPU instructions, the following write-after-write (WAW) dependencies should be avoided if possible. Software can reuse the same target for instructions that execute in the same pipeline without penalty. For example, a Vector/SIMD Multimedia Extension Float instruction can reuse the same target register as a previous Vector/SIMD Multimedia Extension Float immediately without any penalty.

- The target of a Vector/SIMD Multimedia Extension Estimate instruction should not be reused within ten dispatch groups.
- The target of a Vector/SIMD Multimedia Extension Float instruction should not be reused within eight dispatch groups.
- The target of a Vector/SIMD Multimedia Extension Complex instruction should not be reused within five dispatch groups.
- An FPU Load should not re-use an FPU Arithmetic target within six dispatch groups.

For Vector/SIMD Multimedia Extension and FPU instructions, there exist write-port-collisions (WPC). In general these are difficult to avoid in a precise fashion other than to avoid dispatch combinations that are close to those listed below.

- A Vector/SIMD Multimedia Extension Simple instruction must stall 1 cycle if a Vector/SIMD Multimedia Extension Complex was issued 5 cycles ago, a Vector/SIMD Multimedia Extension Float was issued 8 cycles ago, or a Vector/SIMD Multimedia Extension Estimate was issued 10 cycles ago.
- A Vector/SIMD Multimedia Extension Complex instruction must stall 1 cycle if a Vector/SIMD Multimedia Extension Float was issued 2 cycles ago, or a Vector/SIMD Multimedia Extension Estimate was issued 4 cycles ago.
- A Vector/SIMD Multimedia Extension Float instruction must stall 1 cycle if a Vector/SIMD Multimedia Extension Estimate was issued 2 cycles ago.
- A Vector/SIMD Multimedia Extension Store instruction must stall 1 cycle if a FPU Store was issued 12 cycles ago.
- An FPU Load instruction must stall 1 cycle if an FPU Arithmetic instruction was issued 6 cycles ago.

Finally, VSU type 1 instructions should avoid writing the same target register as a VSU type 2 instruction within 12 cycles of each other. This prevents a potential 1-cycle stall that can occur if the writes occur in the same cycle.

A.7.4 General Recommendations

The following general recommendations can be used to help improve performance:

- Use the data-cache block touch (cache hint) instructions (**dcbt**).
- Use the cache replacement management facility for controlling the L2 cache and TLB replacement policy based on a class identifier (classID).
- Preload the TLB.
- Use all of the branch-prediction hints given in the branch instructions.

Appendix B. SPU Instruction Set and Intrinsics

The SPE's support the SPU instruction set (ISA), and the C/C++ language extensions for this instruction set provide data types and intrinsics to programmers writing in C or C++ when they use the `spu_intrinsics.h` header file. This section summarizes both the ISA and the extensions. Because the functions performed by the intrinsics are closely related to the assembly-language instructions of the SPU instruction set, this overview may be helpful in understanding the functions of the intrinsics.

The SPU ISA operates primarily on SIMD vector operands, both fixed- and floating-point, with support for some scalar operands. The PPE and the SPE both execute SIMD instructions, but the two processors execute different instruction sets, and programs for the PPE and SPEs must be compiled by different compilers.

For a complete description of the SPU instructions, intrinsics, and their use, see:

- *Synergistic Processor Unit (SPU) Instruction Set Architecture*.
- *SPU C/C++ Language Extensions*.
- *SPU Application Binary Interface Specification.*
- *SPU Assembly Language Specification.*

B.1 SPU Instruction Set

The SPU instruction set uses instructions that are 4 bytes long and word-aligned. It supports 16-byte operand accesses between storage and its 128 vector registers. For a brief overview of the SPU instruction set, including the data types, addressing modes and instruction types, see *Section 3.3* on page 74. The sections that follow summarize key points of the instruction set.

B.1.1 Data Types

Section 3.3.1 on page 74 contains a list of the SPU data types.

B.1.2 Instructions

Table B-1 contains a complete list of SPU instructions, ordered alphabetically. The SPU has two pipelines, named even (pipeline 0) and odd (pipeline 1), into which it can issue and complete up to two instructions per cycle, one in each of the pipelines. The pipeline is identified for each instruction in the table, along with the latency and stall cycles. The latency values include all stall cycles.

A stall is the number of cycles, after completing the execution of an instruction of a given type, before another instruction of the same type can be issued. For example, double-precision floating-point operations have a 6-cycle stall. Therefore, for sequential double-precision floating-point operations, the second operation will be issued at least 7 cycles after the first operation.

The SPU ISA supports only 16-bit multiplies, so 32-bit multiplies are implemented in software using 16-bit multiplies.

For details about the instruction issue rules, see *Section B.1.3* on page 768. For more information about the instructions, see the *Synergistic Processor Unit Instruction Set Architecture* specification and the *SPU Assembly Language Specification*.

Table B-1. SPU Instructions (Page 1 of 7)

Instruction	Description	Pipeline	Latency[1] (cycles)	Stalls (cycles)
a rt, ra, rb	**Add word**.	0	2	0
absdb rt, ra, rb	**Absolute difference of bytes**.	0	4	0
addx rt, ra, rb	**Add word extended**.	0	2	0
ah rt, ra, rb	**Add halfword**.	0	2	0
ahi rt, ra, s10	**Add halfword immediate**.	0	2	0
ai rt, ra, s10	**Add word immediate**.	0	2	0
and rt, ra, rb	**And**.	0	2	0
andbi rt, ra, s10	**And byte immediate**.	0	2	0
andc rt, ra, rb	**And with complement**.	0	2	0
andhi rt, ra, s10	**And halfword immediate**.	0	2	0
andi rt, ra, s10	**And word immediate**.	0	2	0
avgb rt, ra, rb	**Average bytes**.	0	4	0
bg rt, ra, rb	**Borrow generate word**.	0	2	0
bgx rt, ra, rb	**Borrow generate word extended**.	0	2	0
bi ra	**Branch indirect**.	1	4	0
bid ra	**Branch indirect, disable**.	1	4	0
bie ra	**Branch indirect, enable**.	1	4	0
bihnz rt, ra	**Branch indirect if not zero halfword**.	1	4	0
bihnzd rt, ra	**Branch indirect if not zero halfword, disable**.	1	4	0
bihnze rt, ra	**Branch indirect if not zero halfword, enable**.	1	4	0
bihz rt, ra	**Branch indirect if zero halfword**.	1	4	0
bihzd rt, ra	**Branch indirect if zero halfword, disable**.	1	4	0
bihze rt, ra	**Branch indirect if zero halfword, enable**.	1	4	0
binz rt, ra	**Branch indirect if not zero word**.	1	4	0
binzd rt, ra	**Branch indirect if not zero word, disable**.	1	4	0
binze rt, ra	**Branch indirect if not zero word, enable**.	1	4	0
rt, ra	**Branch indirect and set link**.	1	4	0
bisld rt, ra	**Branch indirect and set link, disable**.	1	4	0
bisle rt, ra	**Branch indirect and set link, enable**.	1	4	0
bisled rt, ra	**Branch indirect and set link on external data**.	1	4	0
bisledd rt, ra	**Branch indirect and set link on external data, disable**.	1	4	0
bislede rt, ra	**Branch indirect and set link on external data, enable**.	1	4	0
biz rt, ra	**Branch indirect if zero word**.	1	4	0

Table B-1. SPU Instructions (Page 2 of 7)

Instruction	Description	Pipeline	Latency[1] (cycles)	Stalls (cycles)
bizd rt, ra	**Branch indirect if zero word, disable**.	1	4	0
bize rt, ra	**Branch indirect if zero word, enable**.	1	4	0
br s18	**Branch relative**.	1	4	0
bra s18	**Branch absolute**.	1	4	0
brasl rt, s18	**Branch absolute and set link**.	1	4	0
brhnz rt, s18	**Branch if not zero halfword**.	1	4	0
brhz rt, s18	**Branch if zero halfword**.	1	4	0
brnz rt, s18	**Branch if not zero word**.	1	4	0
brsl rt, s18	**Branch relative and set link**.	1	4	0
brz rt, s18	**Branch if zero word**.	1	4	0
cbd rt, u7(ra)	**Generate controls for byte insertion (d-form)**.	1	4	0
cbx rt, ra, rb	**Generate controls for byte insertion (x-form)**.	1	4	0
cdd rt, u7(ra)	**Generate controls for doubleword insertion (d-form)**.	1	4	0
cdx rt, ra, rb	**Generate controls for doubleword insertion (x-form)**.	1	4	0
ceq rt, ra, rb	**Compare equal word**.	0	2	0
ceqb rt, ra, rb	**Compare equal byte**.	0	2	0
ceqbi rt, ra, s10	**Compare equal byte immediate**.	0	2	0
ceqh rt, ra, rb	**Compare equal halfword**.	0	2	0
ceqhi rt, ra, s10	**Compare equal halfword immediate**.	0	2	0
ceqi rt, ra, s10	**Compare equal word immediate**.	0	2	0
cflts rt, ra, scale7	**Convert floating to signed integer**.	0	7	0
cfltu rt, ra, scale7	**Convert floating to unsigned integer**.	0	7	0
cg rt, ra, rb	**Carry generate word**.	0	2	0
cgt rt, ra, rb	**Compare greater than word**.	0	2	0
cgtb rt, ra, rb	**Compare greater than byte**.	0	2	0
cgtbi rt, ra, s10	**Compare greater than byte immediate**.	0	2	0
cgth rt, ra, rb	**Compare greater than halfword**.	0	2	0
cgthi rt, ra, s10	**Compare greater than halfword immediate**.	0	2	0
cgti rt, ra, s10	**Compare greater than word immediate**.	0	2	0
cgx rt, ra, rb	**Carry generate word extended**.	0	2	0
chd rt, u7(ra)	**Generate controls for halfword insertion (d-form)**.	1	4	0
chx rt, ra, rb	**Generate controls for halfword insertion (x-form)**.	1	4	0
clgt rt, ra, rb	**Compare logical greater than word**.	0	2	0
clgtb rt, ra, rb	**Compare logical greater than byte**.	0	2	0
clgtbi rt, ra, s10	**Compare logical greater than byte immediate**.	0	2	0
clgth rt, ra, rb	**Compare logical greater than halfword**.	0	2	0

Table B-1. SPU Instructions (Page 3 of 7)

Instruction	Description	Pipeline	Latency[1] (cycles)	Stalls (cycles)
clgthi rt, ra, s10	**Compare logical greater than halfword immediate**.	0	2	0
clgti rt, ra, s10	**Compare logical greater than word immediate**.	0	2	0
clz rt, ra	**Count leading zeros**.	0	2	0
cntb rt, ra	**Count ones in bytes**.	0	4	0
csflt rt, ra, scale7	**Convert signed integer to floating**.	0	7	0
cuflt rt, ra, scale7	**Convert unsigned integer to floating**.	0	7	0
cwd rt, u7(ra)	**Generate controls for word insertion (d-form)**.	1	4	0
cwx rt, ra, rb	**Generate controls for word insertion (x-form)**.	1	4	0
dfa rt, ra, rb	**Double-precision floating add**.	0	13	6
dfm rt, ra, rb	**Double-precision floating multiply**.	0	13	6
dfma rt, ra, rb	**Double-precision floating multiply and add**.	0	13	6
dfms rt, ra, rb	**Double-precision floating multiply and subtract**.	0	13	6
dfnma rt, ra, rb	**Double-precision floating negative multiply and add**.	0	13	6
dfnms rt, ra, rb	**Double-precision floating negative multiply and subtract**.	0	13	6
dfs rt, ra, rb	**Double-precision floating subtract**.	0	13	6
dsync	**Synchronize data**.	1	4	0
eqv rt, ra, rb	**Equivalent**.	0	2	0
fa rt, ra, rb	**Floating add**.	0	6	0
fceq rt, ra, rb	**Floating compare equal**.	0	2	0
fcgt rt, ra, rb	**Floating compare greater than**.	0	2	0
fcmeq rt, ra, rb	**Floating compare magnitude equal**.	0	2	0
fcmgt rt, ra, rb	**Floating compare greater than**.	0	2	0
fesd rt, ra	**Floating extend single to double**.	0	13	6
fi rt, ra, rb	**Floating interpolate**.	0	7	0
fm rt, ra, rb	**Floating multiply**.	0	6	0
fma rt, ra, rb, rc	**Floating multiply and add**.	0	6	0
fms rt, ra, rb, rc	**Floating multiply and subtract**.	0	6	0
fnms rt, ra, rb, rc	**Floating negative multiply and subtract**.	0	6	0
frds rt, ra	**Floating round double to single**.	0	13	6
frest rt, ra	**Floating reciprocal estimate**.	1	4	0
frsqest rt, ra	**Floating reciprocal square root estimate**.	1	4	0
fs rt, ra, rb	**Floating subtract**.	0	6	0
fscrrd rt	**Floating-point status control register read**.	0	13	6
fscrwr ra **fscrwr rt, ra**	**Floating-point status control register write**.	0	7	0
fsm rt, ra	**Form select mask for words**.	1	4	0

Table B-1. SPU Instructions (Page 4 of 7)

Instruction	Description	Pipeline	Latency[1] (cycles)	Stalls (cycles)
fsmb rt, ra	**Form select mask for bytes**.	1	4	0
fsmbi rt, u16	**Form select mask for byte immediate**.	1	4	0
fsmh rt, ra	**Form select mask for halfwords**.	1	4	0
gb rt, ra	**Gather bits from words**.	1	4	0
gbb rt, ra	**Gather bits from bytes**.	1	4	0
gbh rt, ra	**Gather bits from halfwords**.	1	4	0
hbr s11, ra	**Hint for branch (r-form)**.	1	15	0
hbra s11, s18	**Hint for branch (a-form)**.	1	15	0
hbrp	**Hint for branch, prefetch (r-form)**.	1	15	0
hbrr s11, s18	**Hint for branch relative**.	1	15	0
heq ra, rb **heq rt, ra, rb**	**Halt if equal**.	0	2	0
heqi ra, s10 **heqi rt, ra, s10**	**Halt if equal immediate**.	0	2	0
hgt ra, rb **hgt rt, ra, rb**	**Halt if greater than**.	0	2	0
hgti ra, s10 **hgti rt, ra, s10**	**Halt if greater than immediate**.	0	2	0
hlgt ra, rb **hlgt rt, ra, rb**	**Halt if logically greater than**.	0	2	0
hlgti ra, s10 **hlgti rt, ra, s10**	**Halt if logically greater than immediate**.	0	2	0
il rt, s16	**Immediate load word**.	0	2	0
ila rt, u18	**Immediate load address**.	0	2	0
ilh rt, u16	**Immediate load halfword**.	0	2	0
ilhu rt, u16	**Immediate load halfword upper**.	0	2	0
iohl rt, u16	**Immediate OR halfword lower**.	0	2	0
iret **iret ra**	**Interrupt return**.	1	4	0
iretd **iretd ra**	**Interrupt return, disable**.	1	4	0
irete **irete ra**	**Interrupt return, enable**.	1	4	0
lnop	**Nop operation (load)**.	1	0	0
lqa rt, s18	**Load quadword (a-form)**.	1	6	0
lqd rt, s14(ra)	**Load quadword (d-form)**.	1	6	0
lqr rt, s18	**Load quadword instruction relative (a-form)**.	1	6	0
lqx rt, ra, rb	**Load quadword (x-form)**.	1	6	0
mfspr rt, spr	**Move from special purpose register**.	1	6	0

Table B-1. SPU Instructions (Page 5 of 7)

Instruction	Description	Pipeline	Latency[1] (cycles)	Stalls (cycles)
mpy rt, ra, rb	**Multiply**.	0	7	0
mpya rt, ra, rb, rc	**Multiply and add**.	0	7	0
mpyh rt, ra, rb	**Multiply high**.	0	7	0
mpyhh rt, ra, rb	**Multiply high high**.	0	7	0
mpyhha rt, ra, rb	**Multiply high high and add**.	0	7	0
mpyhhau rt, ra, rb	**Multiply high high unsigned and add**.	0	7	0
mpyhhu rt, ra, rb	**Multiply high high unsigned**.	0	7	0
mpyi rt, ra, s10	**Multiply immediate**.	0	7	0
mpys rt, ra, rb	**Multiply and shift right**.	0	7	0
mpyu rt, ra, rb	**Multiply unsigned**.	0	7	0
mpyui rt, ra, s10	**Multiply unsigned immediate**.	0	7	0
mtspr spr, ra	**Move to special purpose register**.	1	6	0
nand rt, ra, rb	**Nand**.	0	2	0
nop **nop rt**	**Nop operation (execute)**.	0	0	0
nor rt, ra, rb	**Nor**.	0	2	0
or rt, ra, rb	**Or**.	0	2	0
orbi rt, ra, s10	**Or byte immediate**.	0	2	0
orc rt, ra, rb	**Or with complement**.	0	2	0
orhi rt, ra, s10	**Or halfword immediate**.	0	2	0
ori rt, ra, s10	**Or word immediate**.	0	2	0
orx rt, ra	**Or word across.**	1	4	0
rchcnt rt, ch	**Read channel count**.	1	6	0
rdch rt, ch	**Read channel**.	1	6	0
rot rt, ra, rb	**Rotate word**.	0	4	0
roth rt, ra, rb	**Rotate halfword**.	0	4	0
rothi rt, ra, s7	**Rotate halfword immediate**.	0	4	0
rothm rt, ra, rb	**Rotate and mask halfword**.	0	4	0
rothmi rt, ra, s6	**Rotate and mask halfword immediate**.	0	4	0
roti rt, ra, s7	**Rotate word immediate**.	0	4	0
rotm rt, ra, rb	**Rotate and mask word**.	0	4	0
rotma rt, ra, rb	**Rotate and mask algebraic word**.	0	4	0
rotmah rt, ra, rb	**Rotate and mask algebraic halfword**.	0	4	0
rotmahi rt, ra, s6	**Rotate and mask algebraic halfword immediate**.	0	4	0
rotmai rt, ra, s7	**Rotate and mask algebraic word immediate**.	0	4	0
rotmi rt, ra, s7	**Rotate and mask word immediate**.	0	4	0

Table B-1. SPU Instructions (Page 6 of 7)

Instruction	Description	Pipeline	Latency[1] (cycles)	Stalls (cycles)
rotqbi rt, ra, rb	**Rotate quadword by bits**.	1	4	0
rotqbii rt, ra, u3	**Rotate quadword by bits immediate**.	1	4	0
rotqby rt, ra, rb	**Rotate quadword by bytes**.	1	4	0
rotqbybi rt, ra, rb	**Rotate quadword by bytes from bit shift count**.	1	4	0
rotqbyi rt, ra, s7	**Rotate quadword by bytes immediate**.	1	4	0
rotqmbi rt, ra, rb	**Rotate and mask quadword by bits**.	1	4	0
rotqmbii rt, ra, s3	**Rotate and mask quadword by bits immediate**.	1	4	0
rotqmby rt, ra, rb	**Rotate and mask quadword by bytes**.	1	4	0
rotqmbybi rt, ra, rb	**Rotate and mask quadword by bytes from bit shift count**.	1	4	0
rotqmbyi rt, ra, s6	**Rotate and mask quadword by bytes immediate**.	1	4	0
selb rt, ra, rb, rc	**Select bits**.	0	2	0
sf rt, ra, rb	**Subtract from word**.	0	2	0
sfh rt, ra, rb	**Subtract from halfword**.	0	2	0
sfhi rt, ra, s10	**Subtract from halfword immediate**.	0	2	0
sfi rt, ra, s10	**Subtract from word immediate**.	0	2	0
sfx rt, ra, rb	**Subtract from word extended**.	0	2	0
shl rt, ra, rb	**Shift left word**.	0	4	0
shlh rt, ra, rb	**Shift left halfword**.	0	4	0
shlhi rt, ra, u5	**Shift left halfword immediate**.	0	4	0
shli rt, ra, u6	**Shift left word immediate**.	0	4	0
shlqbi rt, ra, rb	**Shift left quadword by bits**.	1	4	0
shlqbii rt, ra, u3	**Shift left quadword by bits immediate**.	1	4	0
shlqby rt, ra, rb	**Shift left quadword by bytes**.	1	4	0
shlqbybi rt, ra, rb	**Shift left quadword by bytes from bit shift count**.	1	4	0
shlqbyi rt, ra, u5	**Shift left quadword by bytes immediate**.	1	4	0
shufb rt, ra, rb, rc	**Shuffle bytes**.	1	4	0
stop u14	**Stop and signal**.	1	4	0
stopd ra, rb, rc	**Stop and signal with dependencies**.	1	4	0
stqa rt, s18	**Store quadword (a-form)**.	1	6	0
stqd rt, s14(ra)	**Store quadword (d-form)**.	1	6	0
stqr rt, s18	**Store quadword instruction relative (a-form)**.	1	6	0
stqx rt, ra, rb	**Store quadword (x-form)**.	1	6	0
sumb rt, ra, rb	**Sum bytes into halfword**.	0	4	0
sync	**Synchronize**.	1	4	0
syncc	**Synchronize channel**.	1	4	0

Table B-1. SPU Instructions (Page 7 of 7)

Instruction	Description	Pipeline	Latency[1] (cycles)	Stalls (cycles)
wrch ch, ra	**Write channel**.	1	6	0
xor rt, ra, rb	**Xor**.	0	2	0
xorbi rt, ra, s10	**Exclusive or byte immediate**.	0	2	0
xorhi rt, ra, s10	**Exclusive or halfword immediate**.	0	2	0
xori rt, ra, s10	**Exclusive or word immediate**.	0	2	0
xsbh rt, ra	**Extend sign byte to halfword**.	0	2	0
xshw rt, ra	**Extend sign halfword to word**.	0	2	0
xswd rt, ra	**Extend sign word to doubleword**.	0	2	0

1. The latency cycles include all stall cycles.

B.1.3 Fetch and Issue Rules

B.1.3.1 *Fetch*

Instruction fetches load 32 instructions per LS request. Because the LS is single-ported and load and store instruction frequency is likely to drive LS occupancy to high levels, it is important that DMA- and instruction-fetch activity transfer as much useful data as possible in each LS request.

There are three types of instruction fetches: flush-initiated fetches, hint-for branch fetches, and inline (sequential) prefetches. Of these, application software can directly influence hint-for branch fetches and it has some indirect control over inline prefetches.

B.1.3.2 *Issue*

The SPU issues all instructions in program order according to the pipeline assignment. Each instruction is part of a doubleword-aligned instruction pair called a *fetch group*. A fetch group can have one or two valid instructions, but it must be aligned to doubleword boundaries. This means that the first instruction in the fetch group is from an even word address, and the second instruction from an odd word address. The SPU processes fetch groups one at a time, continuing to the next fetch group when the current instruction group becomes empty. An instruction becomes issueable when register dependencies are satisfied and there is no structural hazard (resource conflict) with prior instructions or DMA or ECC activity.

Dual-issue occurs when a fetch group has two issueable instructions in which the first instruction can be executed on the even pipeline and the second instruction can be executed on the odd pipeline. If a fetch group cannot be dual-issued, but the first instruction can be issued, the first instruction is issued to the proper execution pipeline and the second instruction is held until it can be issued. A new fetch group is loaded after both instructions of the current fetch group are issued.

Table B-2 shows how a short program can be issued by the SPU in the absence of resource conflicts and register dependencies. Even-pipeline instructions are shaded, odd-pipeline instructions are not shaded. The program features four cases of pipeline assignment versus instruction

alignment. The eight instructions are issued in seven cycles. Dual-issue occurs only for the first fetch group. The first fetch group consists of an instruction executed by the even pipeline in address x‘0000’ and an instruction executed by the odd pipeline at address x‘0004’.

Table B-2. Instruction Issue Example

Address	Fetch Group in Memory			Instruction Issue		Clock Cycles
	Even Address	Odd Address		Even Pipeline	Odd Pipeline	
x’0000’	1: Even	2: Odd	→ dual issue (no register dependency)	1: Even	2: Odd	1
x’0008’	3: Even	4: Even		3: Even	Penalty	2
x’0010’	5: Odd	6: Odd		4: Even		3
x’0018’	7: Odd	8: Even			5: Odd	4
					6: Odd	5
					7: Odd	6
				8: Even		7

Instruction alignment can be optimized to minimize execution time by inserting NOP and LNOP instructions. For example, *Table B-3* shows how another short program can be scheduled on an SPU. These eight instructions issue in six cycles. This same program can be issued in five cycles if the instruction alignment is improved.

Table B-3. Example of Instruction Issue without NOP and LNOP

Address	Fetch Group in Memory			Instruction Issue		Clock Cycles
	Even Address	Odd Address		Even Pipe	Odd Pipe	
x’0000’	1: Even	2: Odd	→ dual issue (no register dependency)	1: Even	2: Odd	1
x’0008’	3: Odd	4: Even		—	3: Odd	2
x’0010’	5: Odd	6: Even		4: Even	—	3
x’0018’	7: Even	8: Odd		—	5: Odd	4
x’0020’				6: Even	—	5
			→ dual issue (no register dependency)	7: Even	8: Odd	6

Table B-4 on page 770 shows that a NOP added between instructions 2 and 3 maintains dual-issue through the third fetch group, and an added LNOP continues dual-issue for the remainder of the program.

Table B-4. Example of Instruction Issue Using NOP and LNOP

Address	Fetch Group in Memory			Instruction Issue		Clock Cycles
	Even Address	Odd Address		Even Pipe	Odd Pipe	
x'0000'	1: Even	2: Odd	→ dual issue (no register dependency)	1: Even	2: Odd	1
x'0008'	NOP	3: Odd	→ dual issue (no register dependency)	NOP	3: Odd	2
x'0010'	4: Even	5: Odd	→ dual issue (no register dependency)	4: Even	5: Odd	3
x'0018'	6: Even	LNOP	→ dual issue (no register dependency)	6: Even	LNOP	4
x'0020'	7: Even	8: Odd	→ dual issue (no register dependency)	7: Even	8: Odd	5
				—	—	6

Table B-5 shows how the various operations on the even and odd pipelines are related. Instructions are classified in this table as:

- LS: Load and Store
- HB: Branch Hints
- BR: Branch Resolution
- CH: Channel Interface, Special Purpose Registers
- SH: Shuffle
- LNOP: No Operation (load)
- NOP: No Operation (execute)
- SP: Single-Precision Floating-Point
- DP: Double-Precision Floating-Point
- FI: Floating-Point Integer
- BO: Byte Operations
- FX: Simple Fixed-Point
- WS: Word Rotate and Shift

Table B-5. Execution Pipelines and Result Latency

Pipeline	Pipeline Stage/Latency (cycles)									
	M/1	N/2	O/3	P/4	Q/5	R/6	S/7	T	U	V
Even		FX		BO, WS		SP	FI	Pre-writeback	Register file write	Read bypass
Odd				BR, SH		LS, CH		Pre-writeback	Register file write	Read bypass

Each pipeline stage has a unique alphabetical identifier such that a later stage has a letter that occurs later in the alphabet. Execution begins in stage M, where inputs to the execution units are launched and latched. No unit produces a result in the first cycle, but FX-class instruction results become available for use by subsequent instructions in the second cycle. When a result becomes available in a particular cycle, it is available for use by any subsequent instruction executed by any unit. Although shorter-latency instructions make their results available for forwarding earlier, their results are staged out to latency 8 (stage T) before the results are sent to the register file. The longest latency of a pipelined result is seven cycles, which allows seven cycles of result-equalization to commit register-file results in program order.

Register-file reads and writes are both 2-cycle operations. Result data is sent to the register file while the write addresses are decoded during the pre-writeback stage. In the next cycle, the register file is written. One final stage of forwarding, read bypass, is necessary before the written result is available from the register file.

Double-precision instructions are performed as two double-precision operations in 2-way SIMD fashion. However, the SPU is capable of performing only one double-precision operation per cycle. Thus, the SPU executes double-precision instructions by breaking up the SIMD operands and executing the two operations in consecutive instruction slots in the pipeline. Although double-precision instructions have 13-clock-cycle latencies, only the final seven cycles are pipelined. No other instructions are dual-issued with double-precision instructions, and no instructions of any kind are issued for six cycles after a double-precision instruction is issued.

B.1.4 Inline Prefetch and Instruction Runout

For inline (sequential) prefetch, the prefetcher uses empty slots on the LS schedule to perform the fetch, rather than using a specific instruction as in a branch hint. It is possible to prevent prefetch with long sequences of load and store instructions. The prefetcher starts looking for slots when one of the inline prefetch buffer pairs becomes empty. At this point, there should be 16 instructions in the predicted-path buffer and another 32 instructions in the other prefetch buffer. If the SPU sustains dual-issue, there should be 24 cycles before the prefetched text is needed for inline speculation.

Prefetch requests require 15 cycles to load the instruction line buffer. This means that the prefetcher has nine cycles in which it must succeed in securing a slot. If the LS is busy during this interval and the prefetch is not started, instruction runout may occur (that is, there may be no instructions in the pipeline, and the SPU becomes idle). The minimum runout delay to reset the fetch state machine is three cycles if the prefetch was late by only one or two cycles. Prefetches late enough to cause runout for more than three cycles incrementally delay the issue of the next instruction by one cycle for every cycle the prefetch is delayed. As the pipeline drains, loads and stores cease, the prefetcher can schedule the prefetch, and execution resumes.

Long sequences of instructions that access the LS every cycle can cause instruction runout. Instruction runout can be prevented by breaking up these sequences with instructions that do not access the LS. The **hbrp** instruction allows programs to ensure that the cycles needed for instruction fetch are not consumed by DMA transfers. In principle, any instruction that does not use the LS, and that does not dual-issue with an instruction that uses the LS, is sufficient.

The following rules apply:

- The empty slot in the LS schedule may be adjacent to a slot used by a line-read DMA transfer or branch hint, and this prevents the prefetch from using this empty slot.
- The issue-control unit can schedule non-LS instructions during a DMA transfer, but this causes the slot that the schedule intended for prefetch to be lost.

The **hbrp** instruction has a P feature bit that alters the behavior of the **hbr** instruction. When this bit is set, **hbrp** arbitrates for the LS as though it were a hint. However, after it is scheduled, it becomes a **nop** instruction. This leaves a hole wide enough for an instruction prefetch. For more information, see:

- *SPU Assembly Language Specification*
- *SPU Application Binary Interface Specification*

B.2 C/C++ Language Extensions (Intrinsics) for SPU Instructions

An overview of the SPU intrinsics is given in *Section 3.4* on page 76. The sections that follow summarize key points of the intrinsics. For a complete description, see the *SPU C/C++ Language Extensions* document.

B.2.1 Vector Data Types

The C/C++ language extensions define eleven vector data types, as shown in*Table B-6*. These data types are all 128 bits long and contain from 1 to 16 elements per vector. The `qword` data type is a special 16-byte quadword that is only used as input or output for specific intrinsics (see *Section B.2.3* on page 776).

Table B-6. Vector Data Types (Page 1 of 2)

Vector Data Type	Content
vector unsigned char	16 8-bit unsigned chars
vector signed char	16 8-bit signed chars
vector unsigned short	8 16-bit unsigned halfwords
vector signed short	8 16-bit signed halfwords
vector unsigned int	4 32-bit unsigned words
vector signed int	4 32-bit signed words
vector unsigned long long	2 64-bit unsigned doublewords
vector signed long long	2 64-bit signed doublewords

Table B-6. Vector Data Types (Page 2 of 2)

Vector Data Type	Content
vector float	4 32-bit single-precision floating-point numbers
vector double	2 64-bit double-precision floating-point numbers
qword	quadword (16-byte)

To improve code portability, the `spu_intrinsics.h` header file provides `typedefs` for the vector data types, as shown in *Table B-7*. The `typedefs` may be useful when porting code from the PPE to an SPE.

Table B-7. Programmer Typedefs for Vector Data Types

Vector Data Type	Typedef
vector unsigned char	vec_uchar16
vector signed char	vec_char16
vector unsigned short	vec_ushort8
vector signed short	vec_short8
vector unsigned int	vec_uint4
vector signed int	vec_int4
vector unsigned long long	vec_ullong2
vector signed long long	vec_llong2
vector float	vec_float4
vector double	vec_double2

Table B-8 shows the size and alignment of the data types.

Table B-8. Data Type Alignments

Data Type	Size	Alignment
char	1	byte
short	2	halfword
int	4	word
long	4	word/doubleword
long long	8	doubleword
float	4	word
double	8	doubleword
pointer	4	word
vector	16	quadword

The `aligned` attribute is used to align data on particular boundaries, as in:

```
float factor _attribute_((aligned (16)));
```

The variable `factor` is aligned on a quadword boundary.

The `__align_hint` intrinsic helps in two ways:

- Enables compiler analysis and optimization for pointers.
- Provides compilers with extra information needed for auto-SIMDization.

The `__align_hint(ptr, base, offset)` intrinsic tells the compiler that the pointer `ptr` points to data with a base alignment of `base` and with the given `offset`. The `base` must be a power of 2. A `base` of zero implies the pointer has no known alignment. The `offset` must be less than the `base`, or zero.

Use `__align_hint()` with care. If it is used with pointers that are not aligned, data can end up straddling quadword boundaries. If you specify alignment incorrectly, your program might be compiled incorrectly (that is, not as you expect) and you might get incorrect results.

One last caveat: although a JSRE-compliant compiler is required to provide the `__align_hint` intrinsic, the compiler is allowed to ignore these hints.

24.9.0.1 *Vector Element Ordering*

The SPE intrinsics support big-endian data ordering of elements within a vector data type, as shown in *Figure 24-10*.

Figure 24-10. Element Ordering for Vector Types

msb lsb

byte 0	byte 1	byte 2	byte 3	byte 4	byte 5	byte 6	byte 7	byte 8	byte 9	byte 10	byte 11	byte 12	byte 13	byte 14	byte 15
doubleword 0								doubleword 1							
word 0				word 1				word 2				word 3			
halfword 0		halfword 1		halfword 2		halfword 3		halfword 4		halfword 5		halfword 6		halfword 7	
char 0	char 1	char 2	char 3	char 4	char 5	char 6	char 7	char 8	char 9	char 10	char 11	char 12	char 13	char 14	char 15

B.2.2 Vector Literals

A vector literal is written as a parenthesized vector type followed by a curly-braced set of constant expressions, as shown in *Table B-9*. The elements of the vector are initialized to the corresponding expression. Elements for which no expressions are specified default to 0. Vector literals may be used either in initialization statements or as constants in executable statements.

Table B-9. Vector-Literal Format (Page 1 of 2)

Notation	Represents
(vector unsigned char) *{unsigned int, ...}*	A set of 16 unsigned 8-bit quantities.
(vector signed char) *{signed int, ...}*	A set of 16 signed 8-bit quantities.
(vector unsigned short) *{unsigned short, ...}*	A set of 8 unsigned 16-bit quantities.
(vector signed short) *{signed short, ...}*	A set of 8 signed 16-bit quantities.
(vector unsigned int) *{unsigned int, ...}*	A set of 4 unsigned 32-bit quantities.
(vector signed int) *{signed int, ...}*	A set of 4 signed 32-bit quantities.
(vector unsigned long long) *{unsigned long long, ...}*	A set of 2 unsigned 64-bit quantities.

Table B-9. Vector-Literal Format (Page 2 of 2)

Notation	Represents
(vector signed long long) *{signed long long, ...}*	A set of 2 signed 64-bit quantities.
(vector float) *{float, ...}*	A set of 4 32-bit floating-point quantities.
(vector double) *{double, ...}*	A set of 2 64-bit floating-point quantities.

An alternate format may also be supported, as shown in *Table B-10*. This form consists of a parenthesized vector type followed by a parenthesized set of constant expressions.

Table B-10. Alternate Vector-Literal Format (Page 1 of 2)

Notation	Description
(vector unsigned char)(unsigned int)	A set of 16 unsigned 8-bit quantities that all have the value specified by the integer.
(vector unsigned char)(unsigned int, ..., unsigned int)	A set of 16 unsigned 8-bit quantities specified by the 16 integers.
(vector signed char)(signed int)	A set of 16 signed 8-bit quantities that all have the value specified by the integer.
(vector signed char)(signed int, ..., signed int)	A set of 16 signed 8-bit quantities specified by the 16 integers.
(vector unsigned short)(unsigned int)	A set of 8 unsigned 16-bit quantities that all have the value specified by the integer.
(vector unsigned short)(unsigned int, ..., unsigned int)	A set of 8 unsigned 16-bit quantities specified by the 16 integers.
(vector signed short)(signed int)	A set of 8 signed 16-bit quantities that all have the value specified by the integer.
(vector signed short)(signed int, ..., signed int)	A set of 8 signed 16-bit quantities specified by the 16 integers.
(vector unsigned int)(unsigned int)	A set of 4 unsigned 32-bit quantities that all have the value specified by the integer.
(vector unsigned int)(unsigned int, ..., unsigned int)	A set of 4 unsigned 32-bit quantities specified by the 16 integers.
(vector signed int)(signed int)	A set of 4 signed 32-bit quantities that all have the value specified by the integer.
(vector signed int)(signed int, ..., signed int)	A set of 4 signed 32-bit quantities specified by the 16 integers.
(vector unsigned long long)(unsigned long long)	A set of 2 unsigned 64-bit quantities that all have the value specified by the long long integer.
(vector unsigned long long)(unsigned long long, unsigned long long)	A set of 2 unsigned 64-bit quantities specified by the 2 long long integers.
(vector signed long long)(signed long long)	A set of 2 signed 64-bit quantities that all have the value specified by the long long integer.
(vector signed long long)(signed long long, signed long long)	A set of 2 signed 64-bit quantities specified by the 2 long long integers.
(vector float)(float)	A set of 4 32-bit floating-point quantities that all have the value specified by the float.

Table B-10. Alternate Vector-Literal Format (Page 2 of 2)

Notation	Description
(vector float)(float, float, float, float)	A set of 4 32-bit floating-point quantities specified by the 4 floats.
(vector double)(double)	A set of 2 64-bit double-precision quantities that all have the value specified by the double.
(vector double)(double, double)	A set of 2 64-bit quantities specified by the 2 doubles.

B.2.3 Intrinsics

Table B-11 on page 776 lists the generic and composite intrinsics. Generic intrinsics map to one or more assembly-language instructions, as a function of the type of its input parameters. *Built-ins* are a subset of generic intrinsics that map to more than one SPU instruction. All of the generic intrinsics and built-ins are prefixed by the string, `spu_`. For example, the intrinsic that implements the stop assembly instruction is named `spu_stop`. Many generic intrinsics accept scalars as one of their operands. These correspond to intrinsics that map to instructions with immediate values.

Generic intrinsics are provided for all SPU instruction, *except the following*:

- branch.
- branch hint (see *Section 24.3* on page 689)
- interrupt return.
- generate control for insertion (used for scalar stores).
- constant formation.
- nop.
- memory load and store.
- stop and signal with dependencies (stopd).

Composite intrinsics are constructed from a sequence of specific or generic intrinsics. Specific intrinsics are not included in this summary, because they are rarely used.

Table B-11. SPU Intrinsics (Page 1 of 4)

Intrinsic	Description
Constant Formation	
d = spu_splats(a)	Replicate scalar a into all elements of vector d
Conversion	
d = spu_convtf(a, scale)	Convert integer vector to float vector
d = spu_convts(a, scale)	Convert float vector to signed int vector
d = spu_convtu(a, scale)	Convert float vector to unsigned float vector
d = spu_extend(a)	Sign extend vector
d = spu_rountf(a)	Round double vector to float vector
Arithmetic	
d = spu_add(a, b)	Vector add

Table B-11. SPU Intrinsics (Page 2 of 4)

Intrinsic	Description
d = spu_addx(a, b, c)	Vector add extended
d = spu_genb(a, b)	Vector generate borrow
d = spu_genbx(a, b, c)	Vector generate borrow extended
d = spu_genc(a, b)	Vector generate carry
d = spu_gencx(a, b, c)	Vector generate carry extended
d = spu_madd(a, b, c)	Vector multiply and add
d = spu_mhhadd(a, b, c)	Vector multiply high high and add
d = spu_msub(a, b, c)	Vector multiply and subtract
d = spu_mul(a, b)	Vector multiply
d = spu_mulh(a, b)	Vector multiply high
d = spu_mulhh(a, b)	Vector multiply high high
d = spu_mulo(a, b)	Vector multiply odd
d = spu_mulsr(a, b)	Vector multiply and shift right
d = spu_nmadd(a, b, c)	Negative vector multiply and add
d = spu_nmsub(a, b, c)	Negative vector multiply and subtract
d = spu_re(a)	Vector floating-point reciprocal estimate
d = spu_rsqrte(a)	Vector floating-point reciprocal square root estimate
d = spu_sub(a, b)	Vector subtract
d = spu_subx(a, b, c)	Vector subtract extended
Byte Operation	
d = spu_absd(a, b)	Vector absolute difference
d = spu_avg(a, b)	Vector average
d = spu_sumb(a, b)	Vector sum bytes into shorts
Compare, Branch, and Halt	
d = spu_bisled(func)	Branch indirect and set link if external data
d = spu_cmpabseq(a, b)	Vector compare absolute equal
d = spu_cmpabsgt(a, b)	Vector compare absolute greater than
d = spu_cmpeq(a, b)	Vector compare equal
d = spu_cmpgt(a, b)	Vector compare greater than
(void) spu_hcmpeq(a, b)	Halt if compare equal
(void) spu_hcmpgt(a, b)	Halt if compare greater than
Bit and Mask	
d = spu_cntb(a)	Vector count ones for bytes
d = spu_cntlz(a)	Vector count leading zeros
d = spu_gather(a)	Gather bits from elements
d = spu_maskb(a)	Form select byte mask
d = spu_maskh(a)	Form select halfword mask

Table B-11. SPU Intrinsics (Page 3 of 4)

Intrinsic	Description
d = spu_maskw(a)	Form select word mask
d = spu_sel(a, b, pattern)	Select bits
d = spu_shuffle(a, b, pattern)	Shuffle bytes of a Vector
Logical	
d = spu_and(a, b)	Vector bit-wise AND
d = spu_andc(a, b)	Vector bit-wise AND with complement
d = spu_eqv(a, b)	Vector bit-wise equivalent
d = spu_nand(a, b)	Vector bit-wise complement of AND
d = spu_nor(a, b)	Vector bit-wise complement of OR
d = spu_or(a, b)	Vector bit-wise OR
d = spu_orc(a, b)	Vector bit-wise OR with complement
d = spu_orx(a)	Bit-wise OR word elements
d = spu_xor(a, b)	Vector bit-wise exclusive OR
Rotate	
d = spu_rl(a, count)	Element-wise bit rotate left
d = spu_rlmask(a, count)	Element-wise bit rotate left and mask
d = spu_rlmaska(a, count)	Element-wise bit algebraic rotate and mask
d = spu_rlmaskqw(a, count)	Bit rotate and mask quadword
d = spu_rlmaskqwbyte(a, count)	Byte rotate and mask quadword
d = spu_rlmaskqwbytebc(a, count)	Byte rotate and mask quadword using bit rotate count
d = spu_rlqw(a, count)	Bit rotate quadword left
d = spu_rlqwbyte(a, count)	Byte rotate quadword left
d = spu_rlqwbytebc(a, count)	Byte rotate quadword left using bit rotate count
Shift	
d = spu_sl(a, count)	Element-wise bit shift left
d = spu_slqw(a, count)	Bit shift quadword left
d = spu_slqwbyte(a, count)	Byte shift quadword left
d = spu_slqwbytebc(a, count)	Byte shift quadword left using bit shift count
Control	
(void) spu_idisable()	Disable interrupts
(void) spu_ienable()	Enable interrupts
(void) spu_mffpscr()	Move from floating-point status and control register
(void) spu_mfspr(register)	Move from special purpose register
(void) spu_mtfpscr(a)	Move to floating-point status and control register
(void) spu_mtspr(register, a)	Move to special purpose register
(void) spu_dsync()	Synchronize data
(void) spu_stop(type)	Stop and signal

Table B-11. SPU Intrinsics (Page 4 of 4)

Intrinsic	Description
(void) spu_sync()	Synchronize
Scalar	
d = spu_extract(a, element)	Extract vector element from vector
d = spu_insert(a, b, element)	Insert scalar into specified Vector element
d = spu_promote(a, element)	Promote scalar to vector
Channel Control	
d = spu_readch(channel)	Read word channel
d = spu_readchqw(channel)	Read quadword channel
d = spu_readchcnt(channel)	Read channel count
(void) spu_writech(channel, a)	Write word channel
(void) spu_writechqw(channel, a)	Write quadword channel
Composit Intrinsics	
spu_mfcdma32(ls, ea, size, tagid, cmd)	Initiate DMA to or from 32-bit effective address
spu_mfcdma64(ls, eahi, ealow, size, tagid, cmd)	Initiate DMA to or from 64-bit effective address
spu_mfcstat(type)	Read MFC tag status

For further information about the SPU intrinsics, see the *SPU C/C++ Language Extensions*.

B.2.4 Inline Assembly

Occasionally, the C/C++ language constructs and intrinsic may not be sufficient to achieve a desired low-level programming result. In these situations, inline assembly instructions can be used. The inline assembly syntax must match the AT&T assembly syntax implemented by gcc. The `.balignl` directive may be used within the inline assembly to ensure the known alignment that is needed to achieve effective dual-issue by the hardware.

B.2.5 Compiler Directives

Like compiler intrinsics, a compiler may provide directives that are very useful for SPE programming. For example, the `restrict` qualifier is well-known in many C/C++ implementations, and it is part of the *SPU C/C++ Language Extensions*. When the `restrict` keyword is used to qualify a pointer, it specifies that all accesses to the object pointed to are done through the pointer. For example:

```
void *memcpy(void * restrict s1, void * restrict s2, size_t n);
```

By specifying `s1` and `s2` as pointers that are restricted, the programmer is specifying that the source and destination objects (for the memory copy) do not overlap.

Another useful directive is `__builtin_expect`. Since branch mispredicts are relatively expensive, `__builtin_expect` provides a way for the programmer to direct branch prediction. This example:

```
int __builtin_expect(int exp, int value)
```

returns the result of evaluating `exp`, and means that the programmer expects `exp` to equal `value`. The `value` can be a constant for compile-time prediction, or a variable used for run-time prediction.

Two more useful directives are the `aligned` attribute, and the `_align_hint` directive. The `aligned` attribute is used to ensure proper DMA alignment, for efficient data transfer. The syntax is the same as in many implementations of gcc:

```
float factor __attribute__((aligned (16)); //aligns "factor" to a quadword
```

The `_align_hint` directive helps compilers auto-vectorize. Although it looks like an intrinsic, it is more properly described as a compiler directive, since no code is generated as a result of using the directive. The example

```
_align_hint(ptr, base, offset)
```

can inform the compiler that the pointer, `ptr`, points to data with a base alignment of `base`, with a byte offset from the base alignment of `offset`. The base alignment must be a power of two. Giving 0 as the base alignment implies that the pointer has no known alignment. The offset must be less than the base, or, zero. The `_align_hint` directive should not be used with pointers that are not naturally aligned.

Appendix C. Performance Monitor Signals

The CBE performance-monitoring facilities can monitor many types of events (also called signals) from all major units on the CBE processor. For an introduction to the Performance Monitor Signals, see *Section 16 Performance Monitoring* on page 437.

C.1 Selecting Performance Monitor Signals on the CBE Debug Bus

The performance monitor signals are routed on a 128-bit-wide debug bus to the performance monitor registers in the pervasive logic. To enable the signals from an island to reach the debug bus, one or more Privilege 1 (hypervisor mode) implementation-specific registers within the unit needs to be set up correctly. By default, a value of zero in the affected bit positions in these registers disables the unit from routing its signals to the debug bus. Any unit that needs to be set up for signal routing must have the appropriate registers explicitly initialized by writing the non-zero values as specified in this document. It is expected that the original values will be written back to the affected registers once the performance monitoring of the specific island is complete.

The performance monitor can be used to count the number of occurrences of an event (event-counting) or the number of cycles that a signal is active or inactive (cycle-counting) or, in some cases, both. Event counting is done using the rising edge of a signal, while cycle counting is done using the positive-high level of the signal. Also note that the cycles refer to the clock rate at which the island operates. It is either the core clock rate (NClk) or half the core clock rate (NClk/2).

In the signal group tables that follow, each signal is designated as a type B, C, E, V, or S. These types, which define how the signal needs to be interpreted, are described below:

- **B**—A signal that counts both the number of occurrences and their cumulative length. This signal can present either a single-cycle event or a multi-cycle event. For this signal type, the rising or falling edge of the signal indicates an occurrence of the event, while the level signal denotes the length of the occurrence. This signal type will have any two successive occurrences separated by at least one cycle.
- **C**—A signal that counts only the cumulative length of all occurrences. A level-signal detector is used to count this signal type. Two successive occurrences of this signal type can occur in back-to-back cycles.
- **E**—A signal that counts only the occurrences. An edge transition detector is needed to count the number of times this event occurs. Two successive events of this type will be separated by at least one idle cycle.
- **V**—A signal that lasts for only one cycle even though it represents multi-cycle events. This signal type is useful for counting the number of occurrences, but not the length for which they last.
- **S**—A signal that counts single-cycle event occurrences. Due to the single-cycle nature, the length and the number of occurrences are synonymous. Therefore, counting cycles is sufficient to determine the number of occurrences.

The *count_cycles* field of the performance monitor *pmN_control* registers must be explicitly programmed to count either cycles or edges or both, as shown in *Table C-1* on page 782.

Table C-1. count_cycles Field of the PMx_control Register

Tag	Cycle Counting	Event Counting
B	Cycle	Edge
C	Cycle	n/a
E	n/a	Edge
V	n/a	Cycle
S	Cycle	Cycle

All MMIO addresses referred to in this document are relative to the `BE_MMIO_Base` address. Some of the registers must be altered with a read-modify-write operation. These are designated “rmw” in the register descriptions.

Note: *The registers listed in this appendix are solely for the setup and use of the performance monitor facility. No further description or detail of these registers is documented.*

The CBE chip consists of several units, also referred to as islands. These include the pervasive unit (PRV), the PowerPC Processor Element (PPE), the Synergistic Processor Elements (SPEs), the Element Interconnect Bus (EIB), the Broadband Engine Interface (BEI) Unit, and the Memory Interface Controller (MIC). The PPE consists of two subunits: PowerPC Processor Unit (PPU), and the PowerPC Storage Subsystem (PPSS). Similarly, each SPE consists of a Synergistic Processing Unit (SPU) and a Memory Flow Controller (MFC). The EIB also contains the Token Manager (TKM). Finally, the BEI consists of the BIF Controller (BIC) and the I/O Controller (IOC). All islands clock at NClk/2 except for the PPU, SPU, and parts of the PPSS, all of which clock at NClk.

Each island can logically OR their performance monitor signals onto the debug bus, allowing the monitoring of multiple islands at the same time. The 128-bit-wide debug bus is logically partitioned into four 32-bit-wide lanes. An island may be capable of routing the signals on one or more of the debug bus lanes. The lane on which the signals are routed is selected by initializing the SPRs or MMIO registers in that unit. Care must be taken *not* to allow more than one island to use the *same* lane on the debug bus at the same time.

Table C-2 shows the mapping between the words selected for each island to the debug bus lane on which they appear. It shows which of the debug bus lanes will be busy when an island is chosen to route their performance monitor signals. Again, care must be taken not to allow any island to route their signals on a lane used by another island at the same time.

In general, any island’s word 0 can be selected with any other island’s word 2 for performance monitor counting. Note also that an NClk signal group (PPU, SPU, or PPSS-NClk) can be selected in combination with any NClk/2 signal group as long as they do not use the same lane on the debug bus.

Table C-2. Lanes Used by Different Islands on the Debug Bus (Page 1 of 2)

Debug Bus Lane 0	Debug Bus Lane 1	Debug Bus Lane 2	Debug Bus Lane 3
PPU word 0	PPU word 1		
PPSS-NClk word 0	PPSS-NClk word 1		
PPSS-NClk/2 word 0	PPSS-NClk/2 word 1	PPSS-NClk/2 word 2	PPSS-NClk/2 word 3

Table C-2. Lanes Used by Different Islands on the Debug Bus (Page 2 of 2)

Debug Bus Lane 0	Debug Bus Lane 1	Debug Bus Lane 2	Debug Bus Lane 3
MIC word 0	MIC word 1	MIC word 2	MIC word 3
EIB word 0	EIB word 1	EIB word 2	EIB word 3
BEI word 0	BEI word 1	BEI word 2	BEI word 3
MFC word 0	MFC word 1	MFC word 2	MFC word 3
SPU word 0 phase A	SPU word 0 phase B	SPU word 1 phase A	SPU word 1 phase B

To use the performance monitor, the signals must pass through all the intervening islands to the pervasive logic where the performance monitor logic is located. This is done by activating the debug bus at *all* the units as described in *Table C-3*. All address offsets are relative to the `BE_MMIO_Base` address.

Table C-3. Register Settings to Enable Pass-Through of Debug Bus Signals Through Islands

Address Offset	Register Short Name	Enable Pass-Through
x‘50 0858’	L2_Debug1	rmw bit 61 = ‘1’
x‘50 0958’	CIU_DR1	rmw bit 5 = ‘1’
x‘50 9C98’	on_ramp_trace	rmw bit 39 = ‘1’
x‘50 A010’	MBL_Debug	rmw bit 20 = ‘1’
x‘40 03C8’	SBI_PMCR (MFC0)	rmw bit 38 = ‘1’
x‘40 23C8’	SBI_PMCR (MFC1)	rmw bit 38 = ‘1’
x‘40 43C8’	SBI_PMCR (MFC2)	rmw bit 38 = ‘1’
x‘40 63C8’	SBI_PMCR (MFC3)	rmw bit 38 = ‘1’
x‘40 83C8’	SBI_PMCR (MFC4)	rmw bit 38 = ‘1’
x‘40 A3C8’	SBI_PMCR (MFC5)	rmw bit 38 = ‘1’
x‘40 C3C8’	SBI_PMCR (MFC6)	rmw bit 38 = ‘1’
x‘40 E3C8’	SBI_PMCR (MFC7)	rmw bit 38 = ‘1’

C.1.1 An Example of Setting up the Performance Monitor in PPSS L2 Mode A

A step-by-step description follows of how to set up the performance monitor logic to count the number of hits and misses in the PPSS L2. These events are available in bits 0 and 1, respectively, of the L2 Mode A signal group (see *Section C.3.2 PPSS L2 Cache Controller - Group 1 (NClk/2)* on page 788).

For this example, we use performance monitor counters 0 and 1 in 32-bit mode, and assume that they will not overflow. If there is any possibility of the counters reaching their maximum value, the chosen counters will need to be set up to either stop counting when they reach the maximum value or to use the trace buffer to store their contents periodically, as defined by the *pm_interval* register. Refer to *Cell Broadband Engine Book IV for DD 3.0 and DD 3.1* and *Cell Broadband Engine Registers for DD 3.0 and DD 3.1* for details on how to do either, and for a description of all the registers used in the example below.

1. *Activate the main debug bus through all units.* This step involves doing a read-modify-write to each of the registers shown in *Table C-3*.

2. *Enable performance monitor logic in the L2.* This step requires two read-modify-write operations: one to *L2_Perfmon1* and another to *L2_Debug1* as shown in *Section C.3.2 PPSS L2 Cache Controller - Group 1 (NClk/2)* on page 788. For this example, we will assume that the signals are to be enabled to appear on the debug bus lane 2.
3. *Enable the debug bus for the performance monitor signals from the PPSS L2 island.* This is done by writing the value of x'08000000' to the *debug_bus_control* register.
4. *Route debug bus lane 2 (or word 2) to performance monitor counter input multiplexer bits 0-31*. This is done by writing the value of x'80000000' to the *group_control* register.
5. *Choose the signals to be monitored for performance monitor counters 0 and 1.* The two events we are interested in, L2 hits and L2 misses, are edge-triggered events and appear in bits 0 and 1, respectively. Set up the two performance monitor counters to count these signals by writing x'01400000' to *pm0_control* and x'05400000' to *pm1_control* registers, respectively.
6. *Set the interval timer.* Since the interval timer counts up, write the value x'00000000' to the *pm_interval* register to count up to 2^{32}-1 NClks. In this example, we will assume that the counters will not overflow.
7. *Initialize the counters*. The two counters, pm0_4 and pm1_5, must be initialized by explicitly writing zeros into them.
8. *Start the performance monitor counting logic at the appropriate time.* This is done by writing the value of x'80000000' to the *pm_control* register.
9. *Stop the performance monitor counting logic at the appropriate time.* This is done by writing the value of x'00000000' to the *pm_control* register.

   Note: *The value written to the pm_control register must the same as in step 8, except that bit 0 should be set '0' to disable the counting. For example, if the performance monitor counters are being used in 16-bit mode, the non-zero values in bits 7-10 should be preserved as written in step 8. This step also stops the interval timer.*
10. *Read out the values in the counters.* The 32-bit values in counters *pm0_4* and *pm1_5* will be the number of L2 hits and L2 misses, respectively. The number of NClks that elapsed between the enabling and disabling of the performance monitor logic can be found by reading the *pm_interval* register.

C.2 PowerPC Processing Unit (PPU) Signal Selection

C.2.1 PPU Instruction Unit (NClk)

SPR Address	Register Short Name	Enable Word 0	Enable Word 1
1009	HID1	rmw bits 31, 32:35, 40:41, 48, 58, 60:61 = ‘1 1111 01 0 0 00’	rmw bits 31, 36:39, 40:41, 48, 58, 62:63 = ‘1 1111 01 0 0 01’

Bit	Type	Description
Thread 0		
0	V	Branch instruction committed.
1	E	Branch instruction that caused a misprediction flush is committed. Branch misprediction includes: (1) misprediction of taken or not-taken on conditional branch, (2) misprediction of branch target address on bclr[1] and bcctr[1].
2	C	Instruction buffer empty.
3	E	Instruction effective-address-to-real-address translation (I-ERAT) miss.
4	B	L1 Instruction cache miss cycles. Counts the cycles from the miss event until the returned instruction is dispatched or cancelled due to branch misprediction, completion restart, or exceptions (see **Note 1**).
6	C	Valid instruction available for dispatch, but dispatch is blocked.
9	E	Instruction in pipeline stage EX7 causes a flush.
11	V	Two PowerPC instructions committed. For microcode sequences, only the last microcode operation is counted. Committed instructions are counted two at a time. If only one instruction has committed for a given cycle, this event will not be raised until another instruction has been committed in a future cycle.
Thread 1		
19	V	Branch instruction committed.
20	E	Branch instruction that caused a misprediction flush is committed. Branch misprediction includes: (1) misprediction of taken or not-taken on conditional branch, (2) misprediction of branch target address on bclr[1] and bcctr[1].
21	C	Instruction buffer empty.
22	E	Instruction Effective-address-to-real-address translation (I-ERAT) miss.
23	B	L1 Instruction cache miss cycles. Counts the cycles from the miss event until the returned instruction is dispatched or cancelled due to branch misprediction, completion restart, or exceptions (see **Note 1**).
25	C	Valid instruction available for dispatch, but dispatch is blocked.
28	E	Instruction in pipeline stage EX7 causes a flush.
30	V	Two PowerPC instructions committed. For microcode sequences only, the last microcode operation is counted. Committed instructions are counted two at a time. If only one instruction has committed for a given cycle, this event will not be raised until another instruction has been committed in a future cycle.

1. Counting the rising edge of this event is only an approximation of the number of I-cache misses. If counting edges, note that the rising edge will not occur when an older I-cache miss is being resolved while a new I-cache miss occurs. This case can happen when the first miss is speculative and is canceled on a later cycle while a new non-speculative miss occurs on the redirect fetch address.

C.2.2 PPU Execution Unit (NClk)

SPR Address	Register Short Name	Enable Word 0	Enable Word 1
1009	HID1	rmw bits 31, 32:35, 42:44, 49, 58, 60:61 = ‘1 1111 001 0 0 10’	rmw bits 31, 36:39, 45:47, 49, 58, 62:63 = ‘1 1111 011 0 0 10’

Bit	Type	Description
Thread 0		
2	S	Data effective-address-to-real-address translation (D-ERAT) miss. Not speculative.
3	S	Store request counted at the L2 interface. Counts microcoded PPE sequences more than once (see **Note 1**).
4	S	Load valid at a particular pipe stage. Speculative, since flushed operations are counted as well. Counts microcoded PPE sequences more than once. Misaligned flushes might be counted the first time as well. Load operations include all loads that read data from the cache, **dcbt** and **dcbtst**. Does not include load Vector/SIMD multimedia extension pattern instructions.
5	S	L1 D-cache load miss. Pulsed when there is a miss request that has a tag miss but not an ERAT miss. Speculative, since flushed operations are counted as well.
Thread 1		
18	S	Data effective-address-to-real-address-translation (D-ERAT) miss. Not speculative.
20	S	Load valid at a particular pipe stage. Speculative, since flushed operations are counted as well. Counts microcoded PPE sequences more than once. Misaligned flushes might be counted the first time as well. Load operations include all loads that read data from the cache, **dcbt** and **dcbtst**. Does not include load Vector/SIMD multimedia extension pattern instructions.
21	S	L1 D-cache load miss. Pulsed when there is a miss request that has a tag miss but not an ERAT miss. Speculative, since flushed operations are counted as well.

1. In single-threaded mode, bit 3 will count the stores for Thread 0 or Thread 1, whichever is running. In multithreaded mode, bit 3 may be incorrect.

C.3 PowerPC Storage Subsystem (PPSS) Signal Selection

The address offset of each setup register is relative to the `BE_MMIO_Base` address.

C.3.1 PPSS Bus Interface Unit (NClk/2)

Address Offset	Register Short Name	Enable Word 2
x'50 0B58'	BIU_DbgPerfmon	rmw bits 0:2 = '110'
x'50 0858'	L2_Debug1	rmw bit 61 = '1'

Bit	Type	Description
0	E	Load from MFC memory-mapped I/O (MMIO) space.
1	E	Stores to MFC MMIO space.
6	E	Request token type (even memory bank numbers 0-14).
15	E	Receive 8-beat data from the Element Interconnect Bus (EIB).
16	E	Send 8-beat data to the EIB.
17	E	Send a command to the EIB (includes retried commands).
18	C	Cycles between data request and data grant.
23	C	Non-Cacheable Unit (NCU) Store Command queue not empty (five entries).

C.3.2 PPSS L2 Cache Controller - Group 1 (NClk/2)

Address Offset	Register Short Name	Enable Word 0	Enable Word 2
x‘50 0838’	L2_Perfmon1	rmw bits 59:63 = ‘10001’	
x‘50 0858’	L2_Debug1	rmw bits 43:61, 62 = ‘00000000000000000001 1’	rmw bits 43:61, 63 = ‘00000000000000000001 1’

Bit	Type	Description
0	E	Hit (core interface unit [CIU] loads and stores).
1	E	Miss (CIU loads and stores).
4	E	CIU load miss.
5	E	CIU store to Invalid state (miss).
7	E	Load word and reserve indexed (**lwarx**/**ldarx**) for Thread 0 hits Invalid (I) cache state.
14	E	Store word conditional indexed (**stwcx**/**stdcx**) for Thread 0 hits Invalid (I) cache state when reservation is set (see **Note 1**).
25	C	All four snoop state machines busy.

C.3.3 PPSS L2 Cache Controller - Group 2 (NClk/2)

Address Offset	Register Short Name	Enable Word 0	Enable Word 2
x'50 0838'	L2_Perfmon1	rmw bits 59:63 = '01001'	
x'50 0858'	L2_Debug1	rmw bits 43:61, 62 = '0000000000000000001 1'	rmw bits 43:61, 63 = '0000000000000000001 1'

Bit	Type	Description
4	E	Data line claim (**dclaim**) that received good combined response; includes store/**stcx**/**dcbz** to Shared (S), Shared Last (S_L), or Tagged (T) cache state; does not include **dcbz** to Invalid (I) cache state (see **Note 1**).
11	E	**Dclaim** converted into **rwitm;** may still not get to the bus if **stcx** is aborted (see **Note 2**).
12	E	Store to modified (M), modified unsolicited (M_U), or exclusive (E) cache state.
13	C	8-entry store queue (STQ) full.
14	E	Store dispatched to RC machine is acknowledged.
15	V	Gatherable store (type = 00000) received from CIU.
18	E	Snoop push.
19	E	Send intervention from (S_L | E) cache state to a destination within the same CBE chip.
20	E	Send intervention from (M | M_U) cache state to a destination within the same CBE chip.
25	E	Respond with Retry to a snooped request due to one of the following conflicts: read-and-claim state machine (RC) full address, castout (CO) congruence class, snoop (SNP) machine full address, all snoop machines busy, directory lockout, or parity error.
26	E	Respond with Retry to a snooped request because all snoop machines are busy.
27	E	Snooped response causes a cache state transition from (M | MU) to (E | S | T).
28	E	Snooped response causes a cache state transition from E to S.
29	E	Snooped response causes a cache state transition from (E | SL | S | T) to Invalid (I).
30	E	Snooped response causes a cache state transition from (M | MU) to I.

1. A combined response of Retry for an atomic **dclaim** (due to a lost reservation) is valid.
2. An atomic **dclaim** that is converted to **rwitm** may not get to the bus because of a lost reservation, but will still be counted.

C.3.4 PPSS L2 Cache Controller - Group 3 (NClk/2)

Address Offset	Register Short Name	Enable Word 0	Enable Word 2
x‘50 0838’	L2_Perfmon1	rmw bits 59:63 = ‘00101’	
x‘50 0858’	L2_Debug1	rmw bits 43:61, 62 = ‘00000000000000000001 1’	rmw bits 43:61, 63 = ‘00000000000000000001 1’

Bit	Type	Description
12	E	Load and reserve indexed (**lwarx**/**ldarx**) for Thread 1 hits Invalid (I) cache state.
19	E	Store conditional indexed (**stwcx**/**stdcx**) for Thread 1 hits Invalid (I) cache state when reservation is set.

C.3.5 PPSS Non-Cacheable Unit (NClk/2)

Address Offset	Register Short Name	Enable Word 0
x‘50 0858’	L2_Debug1	rmw bit 61 = ‘1’
x‘50 0A58’	NCU_DR1	rmw bit 0 = ‘1’
x‘50 0A60’	NCU_DR2	rmw bits 0:6 = ‘0000001’

Bit	Type	Description
0	V	Non-cacheable store request received from CIU; includes all synchronization operations such as **sync** and **eieio**.
1	V	**sync** received from CIU.
4	V	Non-cacheable store request received from CIU (includes *only* stores).
6	V	**eieio received** from CIU.
7	V	**tlbie** received from CIU.
8	C	**sync** at the bottom of the store queue (waiting on *st_done* signal from the Bus Interface Unit (BIU) and *sync_done* signal from L2).
9	C	**lwsync** at the bottom of the store queue (waiting for a *sync_done* signal from the L2).
10	C	**eieio** at the bottom of the store queue (waiting for a *st_done* signal from the BIU and a *sync_done* signal from the L2).
11	C	**tlbie** at the bottom of the store queue (waiting for a *st_done* signal from the BIU).
12	V	Non-cacheable store combined with the previous non-cacheable store with a contiguous address.
15	C	All four store-gather buffers full.
16	E	Non-cacheable load request received from CIU (includes instruction and data fetches).
17	C	4-deep store queue not empty.
18	C	4-deep store queue full.
19	C	At least one store gather buffer is not empty.

C.4 Synergistic Processor Unit (SPU) Signal Selection

The address offset of each setup register is relative to the `BE_MMIO_Base` address + (x'2000' * SPE number). For example, the address offset of the *SPU_trace_sel* register in SPU 1 would be at offset x'40_3048' from the `BE_MMIO_Base` address.

C.4.1 SPU (NClk)

Address Offset	Register Short Name	Enable Word 0 (see **Note 1**)	Enable Word 2 (see **Note 1**)
x'40 1048'	SPU_trace_sel	rmw bits 32:63 = x'00001000'	rmw bits 32:63 = x'00000010'
x'40 1070'	SPU_trace_cntl	rmw bit 48 = '1'	
x'40 03C8'	SBI_PMCR	rmw bits 28:38 = '11111111011' (even SPU)	rmw bits 20:27, 36:38 = '11111111 011' (even SPU)
		rmw bits 20:27, 36:38 = '11111111 101' (odd SPU)	rmw bits 28:35, 36:38 = '11111111 101' (odd SPU)

Bit	Type	Description
0	S	Dual instruction committed.
1	S	Single instruction committed.
2	S	Pipeline 0 instruction committed.
3	S	Pipeline 1 instruction committed.
5	B	Local storage busy.
6	S	DMA may conflict with load or store.
7	S	Store instruction to local storage issued.
8	S	Load instruction from local storage issued.
9	S	Floating-Point Unit (FPU) exception.
10	S	Branch instruction committed.
11	S	Non-sequential change of the SPU program counter, which can be caused by branch, asynchronous interrupt, stalled wait on channel, error correction code (ECC) error, and so forth.
12	S	Branch not taken.
13	S	Branch miss prediction; not exact. Certain other code sequences can cause additional pulses on this signal (see **Note 2**).
14	S	Branch hint miss prediction; not exact. Certain other code sequences can cause additional pulses on this signal (see **Note 2**).
15	S	Instruction sequence error.

1. When SPU word 0 is enabled, the speed-converted data is routed from the SPU to debug bus lanes 0 and 1. When SPU word 2 is enabled, the speed-converted data is routed from the SPU to debug bus lanes 2 and 3.
2. Examples of false pulses on mispredict signals include the following:
   - A blocking channel (count = 0) accessed through **wrch** or **rdch** that immediately follows a branch instruction
   - An asynchronous interrupt that occurs after a branch is committed
   - A correctable instruction or data ECC error after a branch is committed but before another instruction is committed
   - If the SPU is stopped through the *SPU_RunCntl* register or by a thermal monitor event after a branch is committed but before another instruction is committed
3. This signal is triggered each time a channel read or channel write instruction is issued to the affected channel and the count for the channel is '0'. Note that a channel read or write may be issued speculatively following a mispredicted branch instruction.

Bit	Type	Description
17	B	Stalled waiting on any blocking channel write (see **Note 3**).
18	B	Stalled waiting on External Event Status (Channel 0) (see **Note 3**).
19	B	Stalled waiting on Signal Notification 1 (Channel 3) (see **Note 3**).
20	B	Stalled waiting on Signal Notification 2 (Channel 4) (see **Note 3**).
21	B	Stalled waiting on DMA Command Opcode or ClassID Register (Channel 21) (see **Note 3**).
22	B	Stalled waiting on Tag Group Status (Channel 24) (see **Note 3**).
23	B	Stalled waiting on List Stall-and-Notify Tag Status (Channel 25) (see **Note 3**).
24	B	Stalled waiting on PPC Core Mailbox (Channel 28) (see **Note 3**).
30	B	Stalled waiting on SPU Mailbox (Channel 29) (see **Note 3**).

1. When SPU word 0 is enabled, the speed-converted data is routed from the SPU to debug bus lanes 0 and 1. When SPU word 2 is enabled, the speed-converted data is routed from the SPU to debug bus lanes 2 and 3.
2. Examples of false pulses on mispredict signals include the following:
   - A blocking channel (count = 0) accessed through **wrch** or **rdch** that immediately follows a branch instruction
   - An asynchronous interrupt that occurs after a branch is committed
   - A correctable instruction or data ECC error after a branch is committed but before another instruction is committed
   - If the SPU is stopped through the *SPU_RunCntl* register or by a thermal monitor event after a branch is committed but before another instruction is committed
3. This signal is triggered each time a channel read or channel write instruction is issued to the affected channel and the count for the channel is ‘0’. Note that a channel read or write may be issued speculatively following a mispredicted branch instruction.

C.4.2 SPU Trigger (NClk)

Enable	Address Offset	Register Short Name	Value
Trigger 0	x‘40 1028’	SPU_trig0_sel	rmw bit 57 = 1
Trigger 1	x‘40 1030’	SPU_trig1_sel	
Trigger 2	x‘40 1038’	SPU_trig2_sel	
Trigger 3	x‘40 1040’	SPU_trig3_sel	
Triggers 0, 1, 2, 3	x‘40 1070’	SPU_trace_cntl	rmw bit 48 = ‘1’
Trigger 0	x‘40 03C8’	SBI_PMCR	rmw bit 39 = ‘1’
Trigger 1			rmw bit 40 = ‘1’
Trigger 2			rmw bit 41 = ‘1’
Trigger 3			rmw bit 42 = ‘1’

Bit	Type	Description
57	B	Stalled waiting on channel operation (see **Notes 1 and 2**).

1. When SPU trigger 0, 1, 2, or 3 is selected, then the speed-converted data will pass through the MFC on trigger bits 4, 5, 6, or 7 of the debug bus, as well as bits 0, 1, 2, or 3.
2. This signal is triggered each time a channel read or channel write instruction is issued to the affected channel and the count for the channel is ‘0’. Note that a channel read or write may be issued speculatively following a mispredicted branch instruction.

C.4.3 SPU Event (NClk)

Enable	Address Offset	Register Short Name	Value
Event 0	x‘40 1050’	SPU_event0_sel	rmw bit 59 = 1
Event 1	x‘40 1058’	SPU_event1_sel	
Event 2	x‘40 1060’	SPU_event2_sel	
Event 3	x‘40 1068’	SPU_event3_sel	
Events 0, 1, 2, 3	x‘40 1070’	SPU_trace_cntl	rmw bit 48 = ‘1’
Event 0	x‘40 03C8’	SBI_PMCR	rmw bit 43 = ‘1’
Event 1			rmw bit 44 = ‘1’
Event 2			rmw bit 45 = ‘1’
Event 3			rmw bit 46 = ‘1’

Bit	Type	Description
59	B	Instruction fetch stall.

Note: When any of the three MIC signal groups is enabled for performance monitoring, Event 0 and Event 1 cannot be used.

C.5 Memory Flow Control (MFC) Signal Selection

The address offset of each setup register is relative to the `BE_MMIO_Base` address + (x'2000 * SPE number). For example, the address of the *ATO_PMCR* register for SPU7 would be at offset x'40_E3D0' from the `BE_MMIO_Base` address.

C.5.1 MFC Atomic Unit (NClk/2)

Address Offset	Register Short Name	Enable Word 0	Enable Word 2
x'40 03D0'	ATO_PMCR	rmw bits 34:35 = '10'	rmw bits 34:35 = '01'
x'40 0580'	SMM_HID	rmw bit 23 = '1'	
x'40 03C8'	SBI_PMCR	rmw bits 28:31, 36:38 = '1111 011' (even SPU)	rmw bits 20:23, 36:38 = '1111 011' (even SPU)
		rmw bits 20:23, 36:38 = '1111 101' (odd SPU)	rmw bits 28:31, 36:38 = '1111 101' (odd SPU)

Bit	Type	Description
1	E	Atomic load received from direct memory access controller (DMAC).
2	E	Atomic **dclaim sent to** synergistic bus interface (SBI)**; includes retried requests.**
3	E	Atomic **rwitm performed sent to SBI; includes retried requests.**
4	E	Atomic load miss caused M_U cache state.
5	E	Atomic load miss caused E cache state.
6	E	Atomic load miss caused S_L cache state.
7	E	Atomic load hits cache.
8	E	Atomic load misses cache with data intervention; sum of signals 4 and 6 above.
14	E	**putllc** or **putlluc** misses cache without data intervention; for **putllc**, counts only when reservation is set for the address.
17	C	Snoop machine busy.
19	E	Snoop caused cache state transition from (M | M_U) to I.
21	E	Snoop caused cache state transition from (E | S | S_L) to I.
23	E	Snoop caused cache state transition from M_U to T.
27	E	Sent modified data intervention to a destination within the same CBE chip.

C.5.2 MFC Direct Memory Access Controller (NClk/2)

Address Offset	Register Short Name	Enable Word 0
x‘40 0880’	DMAC_PMCR	rmw bit 40 = ‘1’
x‘40 03C8’	SBI_PMCR	rmw bits 28:31, 36:38 = ‘1111 011’ (even SPU)
		rmw bits 20:23, 36:38 = ‘1111 101’ (odd SPU)

Bit	Type	Description
0	V	Any flavor of DMA **get[]** command issued to the SBI
1	V	Any flavor of DMA **put[]** command issued to the SBI
2	V	DMA put (**put**) is issued to the SBI; counts only non-zero transfer size cases.
17	V	DMA get data from effective address to local storage (**get**) issued to the SBI; counts only non-zero transfer size cases.

C.5.3 MFC Synergistic Bus Interface (NClk/2)

Address Offset	Register Short Name	Enable Word 0	Enable Word 2
x‘40 03C8’	SBI_PMCR	rmw bits 28:31, 36:38, 47:48, 56 = ‘1111 011 00 1’ (even SPU)	rmw bits 20:23, 36:38, 47:48, 56 = ‘1111 011 01 0’ (even SPU)
		rmw bits 20:23, 36:38, 47:48, 56 = ‘1111 101 00 1’ (odd SPU)	rmw bits 28:31, 36:38, 47:48, 56 = ‘1111 101 01 0’ (odd SPU)

Bit	Type	Description
4	V	Load request sent to element interconnect bus (EIB); includes **read**, **read atomic**, **rwitm, rwitm atomic,** and retried commands
5	V	Store request sent to EIB; includes **wwf, wwc, wwk, dclaim, dclaim atomic,** and retried commands.
6	E	Received data from EIB, including partial cache line data.
7	E	Sent data to EIB, both as a master and a snooper.
8	C	16-deep synergistic bus interface (SBI) queue with outgoing requests not empty; does not include atomic requests.
9	C	16-deep SBI queue with outgoing requests full; does not include atomic requests.
10	V	Sent request to EIB.
12	E	Received data bus grant (includes data sent for MMIO operations).
13	C	Cycles between data bus request and data bus grant.
14	V	Command (read or write) for odd-numbered memory bank; valid only when resource allocation is turned on.
15	V	Command (read or write) for even-numbered memory bank; valid only when resource allocation is turned on.
18	V	Request gets the Retry response (includes local and global requests).
19	E	Sent data bus request to EIB.

C.5.4 MFC Synergistic Memory Management (NClk/2)

Address Offset	Register Short Name	Enable Word 0	Enable Word 2
x‘40 0580‘	SMM_HID	rmw bits 12, 23, 24 = ‘1 1 1’	rmw bits 14, 23, 24 = ‘1 1 1’
x‘40 03C8’	SBI_PMCR	rmw bits 28:31, 36:38 = ‘1111 011’ (even SPU)	rmw bits 20:23, 36:38 = ‘1111 011’ (even SPU)
		rmw bits 20:23, 36:38 = ‘1111 101’ (odd SPU)	rmw bits 28:31, 36:38 = ‘1111 101’ (odd SPU)

Bit	Type	Description
0	E	Translation Lookaside Buffer (TLB) miss without parity or protection errors.
1	C	TLB miss (cycles).
2	E	TLB hit.

C.6 Element Interconnect Bus (EIB) Signal Selection

The address offset of each setup register is relative to the `BE_MMIO_Base` address.

The EIB has four separate ports to the debug bus. They can be used singly or in any combination with each other. The four ports are fed by the global address concentrator (AC0), the local address concentrator (AC1), the data arbiter (DARB), and the token manager (TKM).

C.6.1 EIB Address Concentrator 0 (NClk/2)

Address Offset	Register Short Name	Enable Word 0	Enable Word 2
x‘51 1808‘	EIB_AC0Perf	rmw bits 0:3, 16, 18:21 = ‘1111 0 0000’	rmw bits 8:11, 17, 18:21 = ‘1111 0 0000’

Bit	Type	Description
Group 1		
0	V	Number of **read** and **rwitm** commands (including atomic) AC1 to AC0.
1	V	Number of **dclaim** commands (including atomic) AC1 to AC0.
2	V	Number of **wwk**, **wwc**, and **wwf** commands from AC1 to AC0.
3	V	Number of **sync**, **tlbsync**, and **eieio** commands from AC1 to AC0.
4	V	Number of **tlbie** commands from AC1 to AC0.
11	E	Previous adjacent address match (PAAM) Content Addressable Memory (CAM) hit.
12	E	PAAM CAM miss.
14	V	Command reflected.
Group 2		
16	V	Number of **read** and **rwitm** commands (including atomic) from AC1 to AC0.
17	V	Number of **dclaim** commands (including atomic) from AC1 to AC0.
18	V	Number of **wwk**, **wwc**, and **wwf** commands from AC1 to AC0.
19	V	Number of **sync**, **tlbsync**, and **eieio** commands from AC1 to AC0.
20	V	Number of **tlbie** commands from AC1 to AC0.
27	E	PAAM CAM hit.
28	E	PAAM CAM miss.
30	V	Command reflected.

C.6.2 EIB Address Concentrator 1 (NClk/2)

Address Offset	Register Short Name	Enable Word 0	Enable Word 2
x‘51 1860‘	EIB_DbgL	rmw bits 0:3, 8:11 = ‘0101 1111’	
x‘51 1868’	EIB_DbgR		rmw bits 0:3, 8:11 = ‘0110 1111’

Bit	Type	Description
1	V	Local command from SPE 6.
2	V	Local command from SPE 4.
3	V	Local command from SPE 2.
5	V	Local command from PPE.
6	V	Local command from SPE 1.
7	V	Local command from SPE 3.
8	V	Local command from SPE 5.
9	V	Local command from SPE 7.
12	V	AC1-to-AC0 global command from SPE 6.
13	V	AC1-to-AC0 global command from SPE 4.
14	V	AC1-to-AC0 global command from SPE 2.
15	V	AC1-to-AC0 global command from SPE 0.
16	V	AC1-to-AC0 global command from PPE.
17	V	AC1-to-AC0 global command from SPE 1.
18	V	AC1-to-AC0 global command from SPE 3.
19	V	AC1-to-AC0 global command from SPE 5.
20	V	AC1-to-AC0 global command from SPE 7.
23	V	AC1 sends a global command to AC0.
24	V	AC0 reflects a global command back to AC1.
25	V	AC1 reflects a command back to the bus masters.

C.6.3 EIB Data Ring Arbitrator - Group 1 (NClk/2)

Address Offset	Register Short Name	Enable Word 0	Enable Word 2
x‘51 1860‘	EIB_DbgL	rmw bits 0:3, 8:11 = ‘1110 1111’	
x‘51 1868‘	EIB_DbgR		rmw bits 0:3, 8:11 = ‘1110 1111’

Bit	Type	Description
0	E	Grant on data ring 0.
1	E	Grant on data ring 1.
2	E	Grant on data ring 2.
3	E	Grant on data ring 3.
4	C	Data ring 0 is in use.
5	C	Data ring 1 is in use.
6	C	Data ring 2 is in use.
7	C	Data ring 3 is in use.
8	C	All data rings are idle.
9	C	One data ring is busy.
10	C	Two or three data rings are busy.
11	C	All data rings are busy.
12	B	BIC data request pending.
13	B	SPE 6 data request pending.
14	B	SPE 4 data request pending.
15	B	SPE 2 data request pending.
16	B	SPE 0 data request pending.
17	B	MIC data request pending.
18	B	PPE data request pending.
19	B	SPE 1 data request pending.
20	B	SPE 3 data request pending.
21	B	SPE 5 data request pending.
22	B	SPE 7 data request pending.
24	E	BIC is data destination.
25	E	SPE 6 is data destination.
26	E	SPE 4 is data destination.
27	E	SPE 2 is data destination.
28	E	SPE 0 is data destination.
29	E	MIC is data destination.
30	E	PPE is data destination.
31	E	SPE 1 is data destination.

C.6.4 EIB Data Ring Arbiter - Group 2 (NClk/2)

Address Offset	Register Short Name	Enable Word 0	Enable Word 2
x‘51 1860‘	EIB_DbgL	rmw bits 0:3, 8:11 = ‘1111 1111’	
x‘51 1868‘	EIB_DbgR		rmw bits 0:3, 8:11 = ‘1111 1111’

Bit	Type	Description
0	B	BIC data request pending.
1	B	SPE 6 data request pending.
2	B	SPE 4 data request pending.
3	B	SPE 2 data request pending.
4	B	SPE 0 data request pending.
5	B	MIC data request pending.
6	B	PPE data request pending.
7	B	SPE 1 data request pending.
8	B	SPE 3 data request pending.
9	B	SPE 5 data request pending.
10	B	SPE 7 data request pending.
11	B	IOC data request pending.
12	E	BIC is data destination.
13	E	SPE 6 is data destination.
14	E	SPE 4 is data destination.
15	E	SPE 2 is data destination.
16	E	SPE 0 is data destination.
17	E	MIC is data destination.
18	E	PPE is data destination.
19	E	SPE 1 is data destination.
20	E	SPE3 is data destination.
21	E	SPE 5 is data destination.
22	E	SPE 7 is data destination.
23	E	IOC is data destination.
24	E	Grant on data ring 0.
25	E	Grant on data ring 1.
26	E	Grant on data ring 2.
27	E	Grant on data ring 3.
28	C	All data rings are idle.
29	C	One data ring is busy.
30	C	Two or three data rings are busy.
31	C	All four data rings are busy.

C.6.5 EIB Token Manager (NClk/2)

The address offset of the `TKM_PMCR` register is relative to the `BE_MMIO_Base` address.

The Token Manager (TKM) performance monitor bus is 64 bits wide. The first word is routed to the debug bus lane 0; the second word is routed to the debug bus lane 2.

The TKM performance monitor bus is divided into four half-words (16 bits wide). All signal groups in Group A appear on debug bus 0:15; all in Group B appear on debug bus 16:31; all in Group C appear on debug bus 64:79; and all in Group D appear on debug bus 80:95.

Address Offset	Register Short Name	Enable Word 0	Enable Word 2
x‘50 AFF0’	TKM_PMCR	rmw bits 11, 36, 38:41 = ‘1 1 1111’	rmw bits 11, 37, 46:49 = ‘1 1 1111’
		rmw bits 12:16 = ‘00000’: to disable signal group Ax ‘10000’: to select signal group A0 ‘01000’: to select signal group A1 ‘00100’: to select signal group A2	rmw bits 22:26 = ‘00000’: to disable signal group Cx ‘10000’: to select signal group C0 ‘01000’: to select signal group C1 ‘00100’: to select signal group C2 ‘00010’: to select signal group C3
		rmw bits 17:21 = ‘00000’: to disable signal group Bx ‘10000’: to select signal group B0 ‘01000’: to select signal group B1 ‘00100’: to select signal group B2	rmw bits 27:31 = ‘00000’: to disable signal group Dx ‘10000’: to select signal group D0 ‘01000’: to select signal group D1 ‘00100’: to select signal group D2

C.6.5.1 *EIB Token Manager - Group A0 (NClk/2)*

Address Offset	Register Short Name	Enable Word 0[0:15]
x‘50 AFF0’	TKM_PMCR	rmw bits 11, 12:16, 36, 38:41 = ‘1 10000 1 1111’

Bit	Type	Description
0	S	Even XIO token unused by RAG0.
1	S	Odd XIO token unused by RAG0.
2	S	Even bank token unused by RAG0.
3	S	Odd bank token unused by RAG0.
8	S	Token granted for SPE0.
9	S	Token granted for SPE1.
10	S	Token granted for SPE2.
11	S	Token granted for SPE3.
12	S	Token granted for SPE4.
13	S	Token granted for SPE5.
14	S	Token granted for SPE6.
15	S	Token granted for SPE7.

C.6.5.2 *EIB Token Manager - Group A1 (NClk/2)*

Address Offset	Register Short Name	Enable Word 0[0:15]
x‘50 AFF0’	TKM_PMCR	rmw bits 11, 12:16, 36, 38:41 = ‘1 01000 1 1111’

Bit	Type	Description
0	S	Even XIO token wasted by RAG0; valid only when Unused Enable (UE) = ‘1’ in TKM_CR.
1	S	Odd XIO token wasted by RAG0; valid only when UE = ‘1’ in TKM_CR.
2	S	Even bank token wasted by RAG0; valid only when UE = ‘1’ in TKM_CR.
3	S	Odd bank token wasted by RAG0; valid only when UE = ‘1’ in TKM_CR.
12	S	Even XIO token wasted by RAGU.
13	S	Odd XIO token wasted by RAGU.
14	S	Even bank token wasted by RAGU.
15	S	Odd bank token wasted by RAGU.

C.6.5.3 *EIB Token Manager - Group A2 (NClk/2)*

Address Offset	Register Short Name	Enable Word 0[0:15]
x‘50 AFF0’	TKM_PMCR	rmw bits 11, 12:16, 36, 38:41 = ‘1 00100 1 1111’

Bit	Type	Description
0	S	Even XIO token from RAG0 shared with RAG1.
1	S	Even XIO token from RAG0 shared with RAG2.
2	S	Even XIO token from RAG0 shared with RAG3.
3	S	Odd XIO token from RAG0 shared with RAG1.
4	S	Odd XIO token from RAG0 shared with RAG2.
5	S	Odd XIO token from RAG0 shared with RAG3.
6	S	Even bank token from RAG0 shared with RAG1.
7	S	Even bank token from RAG0 shared with RAG2.
8	S	Even bank token from RAG0 shared with RAG3.
9	S	Odd bank token from RAG0 shared with RAG1.
10	S	Odd bank token from RAG0 shared with RAG2.
11	S	Odd bank token from RAG0 shared with RAG3.

C.6.5.4 *EIB Token Manager - Group B0 (NClk/2)*

Address Offset	Register Short Name	Enable Word 0[16:31]
x‘50 AFF0’	TKM_PMCR	rmw bits 11, 17:21, 36, 38:41 = ‘1 10000 1 1111’

Bit	Type	Description
16	S	Even XIO token unused by RAG1.
17	S	Odd XIO token unused by RAG1.
18	S	Even bank token unused by RAG1.
19	S	Odd bank token unused by RAG1.
25	S	Token granted for IOC0.
26	S	Token granted for IOC1.

C.6.5.5 *EIB Token Manager - Group B1 (NClk/2)*

Address Offset	Register Short Name	Enable Word 0[16:31]
x‘50 AFF0’	TKM_PMCR	rmw bits 11, 17:21, 36, 38:41 = ‘1 01000 1 1111’

Bit	Type	Description
16	S	Even XIO token wasted by RAG1; valid only when UE = ‘1’ in TKM_CR.
17	S	Odd XIO token wasted by RAG1; valid only when UE = ‘1’ in TKM_CR.
18	S	Even bank token wasted by RAG1; valid only when UE = ‘1’ in TKM_CR.
19	S	Odd bank token wasted by RAG1; valid only when UE = ‘1’ in TKM_CR.

C.6.5.6 *EIB Token Manager - Group B2 (NClk/2)*

Address Offset	Register Short Name	Enable Word 0[16:31]
x‘50 AFF0’	TKM_PMCR	rmw bits 11, 17:21, 36, 38:41 = ‘1 00100 1 1111’

Bit	Type	Description
16	S	Even XIO token from RAG1 shared with RAG0.
17	S	Even XIO token from RAG1 shared with RAG2.
18	S	Even XIO token from RAG1 shared with RAG3.
19	S	Odd XIO token from RAG1 shared with RAG0.
20	S	Odd XIO token from RAG1 shared with RAG2.
21	S	Odd XIO token from RAG1 shared with RAG3.
22	S	Even bank token from RAG1 shared with RAG0.
23	S	Even bank token from RAG1 shared with RAG2.
24	S	Even bank token from RAG1 shared with RAG3.
25	S	Odd bank token from RAG1 shared with RAG0.
26	S	Odd bank token from RAG1 shared with RAG2.
27	S	Odd bank token from RAG1 shared with RAG3.
28	S	Even XIO token from RAGU shared with RAG1.

Bit	Type	Description
29	S	Odd XIO token from RAGU shared with RAG1.
30	S	Even bank token from RAGU shared with RAG1.
31	S	Odd bank token from RAGU shared with RAG1.

C.6.5.7 *EIB Token Manager - Group C0 (NClk/2)*

Address Offset	Register Short Name	Enable Word 2[64:79]
x'50 AFF0'	TKM_PMCR	rmw bits 11, 22:26, 37, 46:49 = '1 10000 1 1111'

Bit	Type	Description
64	S	Even XIO token unused by RAG2.
65	S	Odd XIO token unused by RAG2.
66	S	Even bank token unused by RAG2.
67	S	Odd bank token unused by RAG2.
68	S	IOIF0 In token unused by RAG0.
69	S	IOIF0 Out token unused by RAG0.
70	S	IOIF1 In token unused by RAG0.
71	S	IOIF1 Out token unused by RAG0.

C.6.5.8 *EIB Token Manager - Group C1 (NClk/2)*

Address Offset	Register Short Name	Enable Word 2[64:79]
x'50 AFF0'	TKM_PMCR	rmw bits 11, 22:26, 37, 46:49 = '1 01000 1 1111'

Bit	Type	Description
64	S	Even XIO token wasted by RAG2; valid only when UE = '1' in TKM_CR.
65	S	Odd XIO token wasted by RAG2; valid only when UE = '1' in TKM_CR.
66	S	Even bank token wasted by RAG2; valid only when UE = '1' in TKM_CR.
67	S	Odd Bank token wasted by RAG2; valid only when UE = '1' in TKM_CR.

C.6.5.9 *EIB Token Manager - Group C2 (NClk/2)*

Address Offset	Register Short Name	Enable Word 2[64:79]
x'50 AFF0'	TKM_PMCR	rmw bits 11, 22:26, 37, 46:49 = '1 00100 1 1111'

Bit	Type	Description
64	S	Even XIO token from RAG2 shared with RAG0.
65	S	Even XIO token from RAG2 shared with RAG1.

Bit	Type	Description
66	S	Even XIO token from RAG2 shared with RAG3.
67	S	Odd XIO token from RAG2 shared with RAG0.
68	S	Odd XIO token from RAG2 shared with RAG1.
69	S	Odd XIO token from RAG2 shared with RAG3.
70	S	Even bank token from RAG2 shared with RAG0.
71	S	Even bank token from RAG2 shared with RAG1.
72	S	Even bank token from RAG2 shared with RAG3.
73	S	Odd bank token from RAG2 shared with RAG0.
74	S	Odd bank token from RAG2 shared with RAG1.
75	S	Odd bank token from RAG2 shared with RAG3.

C.6.5.10 *EIB Token Manager - Group C3 (NClk/2)*

Address Offset	Register Short Name	Enable Word 2[64:79]
x‘50 AFF0’	TKM_PMCR	rmw bits 11, 22:26, 37, 46:49 = ‘1 00010 1 1111’

Bit	Type	Description
64	S	IOIF0 In token wasted by RAG0; valid only when UE = ‘1’ in TKM_CR.
65	S	IOIF0 Out token wasted by RAG0; valid only when UE = ‘1’ in TKM_CR.
66	S	IOIF1 In token wasted by RAG0; valid only when UE = ‘1’ in TKM_CR.
67	S	IOIF1 Out token wasted by RAG0; valid only when UE = ‘1’ in TKM_CR.

C.6.5.11 *EIB Token Manager - Group D0 (NClk/2)*

Address Offset	Register Short Name	Enable Word 2[80:95]
x‘50 AFF0’	TKM_PMCR	rmw bits 11, 27:31, 37, 46:49 = ‘1 10000 1 1111’

Bit	Type	Description
80	S	Even XIO token unused by RAG3.
81	S	Odd XIO token unused by RAG3.
82	S	Even bank token unused by RAG3.
83	S	Odd bank token unused by RAG3.

C.6.5.12 *EIB Token Manager - Group D1 (NClk/2)*

Address Offset	Register Short Name	Enable Word 2[80:95]
x‘50 AFF0’	TKM_PMCR	rmw bits 11, 27:31, 37, 46:49 = ‘1 01000 1 1111’

Bit	Type	Description
80	S	Even XIO token wasted by RAG3; valid only when UE = ‘1’ in TKM_CR.
81	S	Odd XIO token wasted by RAG3; valid only when UE = ‘1’ in TKM_CR.
82	S	Even bank token wasted by RAG3; valid only when UE = ‘1’ in TKM_CR.
83	S	Odd bank token wasted by RAG3; valid only when UE = ‘1’ in TKM_CR.

C.6.5.13 *EIB Token Manager - Group D2 (NClk/2)*

Address Offset	Register Short Name	Enable Word 2[80:95]
x‘50 AFF0’	TKM_PMCR	rmw bits 11, 27:31, 37, 46:49 = ‘1 00100 1 1111’

Bit	Type	Description
80	S	Even XIO token from RAG3 shared with RAG0.
81	S	Even XIO token from RAG3 shared with RAG1.
82	S	Even XIO token from RAG3 shared with RAG2.
83	S	Odd XIO token from RAG3 shared with RAG0.
84	S	Odd XIO token from RAG3 shared with RAG1.
85	S	Odd XIO token from RAG3 shared with RAG2.
86	S	Even bank token from RAG3 shared with RAG0.
87	S	Even bank token from RAG3 shared with RAG1.
88	S	Even bank token from RAG3 shared with RAG2.
89	S	Odd bank token from RAG3 shared with RAG0.
90	S	Odd bank token from RAG3 shared with RAG1.
91	S	Odd bank token from RAG3 shared with RAG2.

C.7 Memory Interface Controller (MIC) Signal Selection

The address offset of each setup register is relative to the `BE_MMIO_Base` address.

C.7.1 MIC Group 1 (NClk/2)

Address Offset	Register Short Name	Enable Words 0 and 1	Enable Words 2 and 3
x'50 A238'	MIC_FIR_Debug	rmw bits 32:33 = '10'	rmw bits 32:33 = '01'
x'50 A1C8'	MIC_Ctl_Debug_1	write bits 0:63 = x'FFFF_FFFF_FFFF_FFFF'	
x'50 A048'	MIC_Ctl_Debug2	rmw bits 0:31 = x'FFFF_FFFF'	
x'50 A228'	MIC_DF_Debug_Yrac_1	write bits 0:63 = x'FFFF_FFFF_FFFF_FFFF'	

Bit	Type	Description
9	C	XIO1 - Read command queue is empty.
10	C	XIO1 - Write command queue is empty.
12	C	XIO1 - Read command queue is full.
13	S	XIO1 - MIC responds with a Retry for a read command because the read command queue is full.
14	C	XIO1 - Write command queue is full.
15	S	XIO1 - MIC responds with a Retry for a write command because the write command queue is full.
34	S	XIO1 - Read command dispatched (includes high-priority and fast-path reads) (see **Note 2**).
35	S	XIO1 - Write command dispatched (see **Note 2**).
36	S	XIO1 - Read-Modify-Write command (data size < 16 bytes) dispatched (see **Note 2**).
37	S	XIO1 - Refresh dispatched (see **Note 2**).
39	S	XIO1 - Byte-masking write command (data size ≥ 16 bytes) dispatched (see **Note 2**).
41	S	XIO1 - Write command dispatched after a read command was previously dispatched (see **Note 2**).
42	S	XIO1 - Read command dispatched after a write command was previously dispatched (see **Note 2**).

1. While this group is enabled for monitoring, SPU event bit 0 cannot be used.
2. This performance monitor signal originates from the MiClk domain, which is synchronous with the XDR interface. It is reliable only when the system operates at nominal clock frequencies (that is, MiClk ≤ NClk/2). In Slow Mode, for instance, when the NClk/2 is slower than the MiClk, the performance monitor event signal from the MiClk domain will change so frequently that it will not be possible to capture and synchronize all MiClk data for the NClk representation.

C.7.2 MIC Group 2 (NClk/2)

Address Offset	Register Short Name	Enable Words 0 and 1	Enable Words 2 and 3
x‘50 A238‘	MIC_FIR_Debug	rmw bits 32:33 = ‘01’	rmw bits 32:33 = ‘10’
x‘50 A088’	MIC_Ctl_Debug_0	write bits 0:63 = x‘FFFF_FFFF_FFFF_FFFF’	
x‘50 A048’	MIC_Ctl_Debug2	rmw bits 0:31 = x‘FFFF_FFFF’	
x‘50 A220’	MIC_DF_Debug_Yrac_0	write bits 0:63 = x‘0000_0000_0000_0000’	

Bit	Type	Description
9	C	XIO0 - Read command queue is empty.
10	C	XIO0 - Write command queue is empty.
12	C	XIO0 - Read command queue is full.
13	S	XIO0 - MIC responds with a Retry for a read command because the read command queue is full.
14	C	XIO0 - Write command queue is full.
15	S	XIO0 - MIC responds with a Retry for a write command because the write command queue is full.
34	S	XIO0 - Read command dispatched (includes high-priority and fast-path reads) (see **Note 2**).
35	S	XIO0 - Write command dispatched (see **Note 2**).
36	S	XIO0 - Read-Modify-Write command (data size < 16 bytes) dispatched (see **Note 2**).
37	S	XIO0 - Refresh dispatched (see **Note 2**).
41	S	XIO0 - Write command dispatched after a read command was previously dispatched (see **Note 2**).
42	S	XIO0 - Read command dispatched after a write command was previously dispatched (see **Note 2**).

1. While this group is enabled for monitoring, the SPU event bit 1 cannot be used.
2. This performance monitor signal originates from the MiClk domain. It is reliable only when the system operates at nominal clock frequencies (that is, MiClk ≤ NClk/2). In Slow Mode, for instance, when the NClk/2 is slower than the MiClk, the performance monitor event signal from the MiClk domain will change so frequently that it will not be possible to capture and synchronize all MiClk data for the NClk representation.

C.7.3 MIC Group 3 (NClk/2)

Address Offset	Register Short Name	Enable Words 0 and 1	Enable Words 2 and 3
x‘50 A238‘	MIC_FIR_Debug	rmw bits 32:33 = ‘01’	rmw bits 32:33 = ‘10’
x‘50 A088’	MIC_Ctl_Debug_0	write bits 0:63 = x‘FDFF_FFFF_FFFF_FFFF’	
x‘50 A048’	MIC_Ctl_Debug2	rmw bits 0:31 = x‘FDFF_FFFF’	
x‘50 A220’	MIC_DF_Debug_Yrac_0	write bits 0:63 = x‘FFFF_FFFF_FFFF_FFFF’	

Bit	Type	Description
35	S	XIO0 - Write command dispatched (see **Note 2**).
36	S	XIO0 - Read-Modify-Write command (data size < 16 bytes) dispatched (see **Note 2**).
37	S	XIO0 - Refresh dispatched (see **Note 2**).
39	S	XIO0 - Byte-masking write command (data size ≥ 16 bytes) dispatched (see **Note 2**).

1. While this group is enabled for monitoring, SPU event bit 1 cannot be used.
2. This performance monitor signal originates from the MiClk domain. It is reliable only when the system operates at nominal clock frequencies (that is, MiClk ≤ NClk/2). In Slow Mode, for instance, when the NClk/2 is slower than the MiClk, the performance monitor event signal from the MiClk domain will change so frequently that it will not be possible to capture and synchronize all MiClk data for the NClk representation.

C.8 Broadband Engine Interface (BEI) Unit

The address offset of each setup register is relative to the `BE_MMIO_Base` address.

The BIF Controller (BIC) implements two I/O links (IOIF0 and IOIF1), and these two instances make use of the same building blocks. Each BIC building block in IOIF0 and IOIF1 places performance monitor signals onto the same word (word 0 or word 2), but the link outputs can be swapped between word 0 and word 2 controlled by the `IF0TRC0[9:10]` for IOIF0 and IOIF1 as described in the tables below.

All I/O Controller (IOC) signal groups, except Group 3, place their signals on the same word (word 0 or word 2). Their link outputs can be swapped between word 0 and word 2, using the same scheme (shown below) as for the BIC groups. For IOC Group 3, however, the selection of the output word is controlled by byte selects in the `IOC_DTB_Cfg0` register.

IOIF0 Word Swapping

Address Offset	Register Short Name	**Typical** IOIF0 Word 0 → BEI Word 0 IOIF0 Word 2 → BEI Word 2	**Swapped** IOIF0 Word 0 → BEI Word 2 IOIF0 Word 2 → BEI Word 0
x‘51 1000’	IF0TRC0	rmw bits 9:10 = ‘00’	rmw bits 9:10 = ‘11’

IOIF1 Word Swapping

Address Offset	Register Short Name	**Typical** IOIF1 Word 0 → BEI Word 0 IOIF1 Word 2 → BEI Word 2	**Swapped** IOIF1 Word 0 → BEI Word 2 IOIF1 Word 2 → BEI Word 0
x‘51 1400’	IF1TRC0	rmw bits 9:10 = ‘00’	rmw bits 9:10 = ‘11’

IOIF0 Byte Enables (applies to the data after the IOIF0 word swap)

Address Offset	Register Short Name	BEI Word 0	BEI Word 2
x‘51 1000’	IF0TRC0	rmw bits 11:14 = ‘1111’	rmw bits 1:4 = ‘1111’

IOIF1 Byte Enables (applies to the data after the IOIF1 word swap)

Address Offset	Register Short Name	BEI Word 0	BEI Word 2
x‘51 1400’	IF1TRC0	rmw bits 1:4 = ‘1111’	rmw bits 11:14 = ‘1111’

C.8.1 BIF Controller - IOIF0 Word 0 (NClk/2)

Address Offset	Register Short Name	Enable Word 0	Enable Word 2
x‘51 2000‘	IF0TRC2	rmw bits 0, 24:28 = ‘1 01000’	
x‘51 1040’	IF0TRC1	rmw bits 0, 4:7, 35 = ‘1 0100 1’	
x‘50 8600’	IOC_DTB_Cfg0	rmw bit 0 = ‘1’	
x‘51 1400’	IF1TRC0	rmw bit 0 = ‘1’	
x‘51 1000’	IF0TRC0	rmw bits 0, 9:10, 11:14 = ‘1 00 1111’	rmw bits 0, 1:4, 9:10 = ‘1 1111 11’

Address Offset	Register Short Name	Disable Word 0	Disable Word 2
x‘51 2000‘	IF0TRC2	rmw bits 24:28 = ‘10000’	
x‘51 1040’	IF0TRC1	rmw bit 4:7, 35 = ‘1000 0’	

Bit	Type	Description
12	S	Type A data physical layer group (PLG). Does not include header-only or credit-only data PLGs. In IOIF mode, counts I/O device read data; in BIF mode, counts all outbound data.
13	S	Type B data PLG. In IOIF mode, counts I/O device read data; in BIF mode, counts all outbound data.
14	S	Type A data PLG. Does not include header-only or credit-only PLGs. In IOIF mode, counts CBE store data to I/O device. Does not apply in BIF mode.
15	S	Type B data PLG. In IOIF mode, counts CBE store data to an I/O device. Does not apply in BIF mode.
16	S	Data PLG. Does not include header-only or credit-only PLGs.
17	S	Command PLG (no credit-only PLG). In IOIF mode, counts I/O command or reply PLGs. In BIF mode, counts command/reflected command or snoop/combined responses.
18	S	Type A data transfer regardless of length. Can also be used to count Type A data header PLGs (but not credit-only PLGs).
19	S	Type B data transfer.
20	S	Command-credit-only command PLG in either IOIF or BIF mode.
21	S	Data-credit-only data PLG sent in either IOIF or BIF mode.
22	S	Non-null envelope sent (does not include long envelopes).
24	E	Null envelope sent (see **Note**).
25	V	No valid data sent this cycle (see **Note**).
26	E	Normal envelope sent (see **Note**).
27	E	Long envelope sent (see **Note**).
28	V	A Null PLG inserted in an outgoing envelope.
29	C	Outbound envelope array is full.

Note: This signal comes from the BClk domain, which is synchronous with the FlexIO interface. Counting back-to-back cycles in the BClk domain may be inaccurate since the signal crosses into the NClk domain to reach the performance monitor counters.

A performance monitor signal from the BClk domain is reliable only when the NClk domain operates at more than two times the BClk domain (that is, NClk is greater than 3.33 GHz).

C.8.2 BIF Controller - IOIF1 Word 0 (NClk/2)

Address Offset	Register Short Name	Enable Word 0	Enable Word 2
x‘51 3000‘	IF1TRC2	rmw bits 0, 24:28 = ‘1 01000’	
x‘51 1440’	IF1TRC1	rmw bits 0, 4:7, 35 = ‘1 0100 1’	
x‘50 8600’	IOC_DTB_Cfg0	rmw bit 0 = ‘1’	
x‘51 1000’	IF0TRC0	rmw bit 0 = ‘1’	
x‘51 1400’	IF1TRC0	rmw bits 0, 1:4, 9:10 = ‘1 1111 00’	rmw bits 0, 9:10, 11:14 = ‘1 11 1111’

Address Offset	Register Short Name	Disable Word 0	Disable Word 2
x‘51 3000‘	IF1TRC2	rmw bits 24:28 = ‘10000’	
x‘51 1440’	IF1TRC1	rmw bit 4:7, 35 = ‘1000 0’	

Bit	Type	Description
19	S	Type B data transfer.

C.8.3 BIF Controller - IOIF0 Word 2 (NClk/2)

Address Offset	Register Short Name	Enable Word 0	Enable Word 2
x‘51 2000‘	IF0TRC2	rmw bits 0, 32:36 = ‘1 01000’	
x‘51 1040’	IF0TRC1	rmw bits 0, 17:20, 25:27, 35 = ‘1 0100 010 1’	
x‘50 8600’	IOC_DTB_Cfg0	rmw bit 0 = ‘1’	
x‘51 1400’	IF1TRC0	rmw bit 0 = ‘1’	
x‘51 1000’	IF0TRC0	rmw bits 0, 9:10, 11:14, 43:45 = ‘1 11 1111 010’	rmw bits 0, 1:4, 9:10, 43:45 = ‘1 1111 00 010’

Address Offset	Register Short Name	Disable Word 0	Disable Word 2
x‘51 2000‘	IF0TRC2	rmw bits 32:36 = ‘10000’	
x‘51 1040’	IF0TRC1	rmw bits 17:20, 25:27, 35 = ‘1000 100 0’	
x‘51 1000’	IF0TRC0	rmw bits 43:45 = ‘100’	

Bit	Type	Description
1	E	Null envelope received (see **Note**).
14	S	Command PLG (no credit-only PLG). In IOIF mode, counts I/O command or reply PLGs. In BIF mode, counts command/reflected command or snoop/combined responses.
15	S	Command-credit-only command PLG.
20	E	Normal envelope received is good (see **Note**).
21	E	Long envelope received is good (see **Note**).
22	S	Data-credit-only data PLG in either IOIF or BIF mode (will count a maximum of one per envelope) (see **Note**).
23	S	Non-null envelope (does not include long envelopes; includes retried envelopes) (see **Note**).
24	S	Data grant received.
28	S	Data PLG. Does not include header-only or credit-only PLGs.
29	S	Type A data transfer regardless of length. Can also be used to count Type A data header PLGs (but not credit-only PLGs).
30	S	Type B data transfer.

Note: This signal comes from the BClk domain. Counting back-to-back cycles in the BClk domain may be inaccurate since the signal crosses into the NClk domain to reach the performance monitor counters.

A performance monitor signal from the BClk domain is reliable only when the NClk domain operates at more than two times the BClk domain (that is, NClk is greater than 3.33 GHz).

C.8.4 BIF Controller - IOIF1 Word 2 (NClk/2)

Address Offset	Register Short Name	Enable Word 0	Enable Word 2
x‘51 3000‘	IF1TRC2	rmw bits 0, 32:36 = ‘1 01000’	
x‘51 1440’	IF1TRC1	rmw bits 0, 17:20, 25:27, 35 = ‘1 0100 010 1’	
x‘50 8600’	IOC_DTB_Cfg0	rmw bit 0 = ‘1’	
x‘51 1000’	IF0TRC0	rmw bit 0 = ‘1’	
x‘51 1400’	IF1TRC0	rmw bits 0, 1:4, 9:10, 43:45 = ‘1 1111 11 010’	rmw bits 0, 9:10, 11:14, 43:45 = ‘1 00 1111 010’

Address Offset	Register Short Name	Disable Word 0	Disable Word 2
x‘51 3000‘	IF1TRC2	rmw bits 32:36 = ‘10000’	
x‘51 1440’	IF1TRC1	rmw bits 17:20, 25:27, 35 = ‘1000 100 0’	
x‘51 1400’	IF1TRC0	rmw bits 43:45 = ‘100’	

Bit	Type	Description
1	E	Null envelope received (see **Note 1**).
14	S	Command PLG (no credit-only PLG). Counts I/O command or reply PLGs.
15	S	Command-credit-only command PLG.
20	E	Normal envelope received is good (see **Note 1**).
21	E	Long envelope received is good (see **Note 1**).
22	S	Data-credit-only data PLG received (will count a maximum of one per envelope) (see **Note 1**).
23	S	Non-Null envelope received (does not include long envelopes; includes retried envelopes) (see **Note 1**).
24	S	Data grant received.
28	S	Data PLG received. Does not include header-only or credit-only PLGs.
29	S	Type A data transfer regardless of length. Can also be used to count Type A data header PLGs (but not credit-only PLGs).
30	S	Type B data transfer received.

1. This signal comes from the BClk domain. Counting back-to-back cycles in the BClk domain can sometimes be inaccurate since the signal crosses into the NClk domain to reach the performance monitor counters. A performance monitor signal from the BClk domain is reliable only when the NClk domain operates at more than two times the BClk domain (that is, NClk is greater than 3.33 GHz).

C.8.5 I/O Controller Word 0 - Group 1 (NClk/2)

Address Offset	Register Short Name	Enable Word 0	Enable Word 2
x‘50 8600’	IOC_DTB_Cfg0	rmw bits 0,16:19 = ‘1 1111’	
x‘51 1C70‘	IOC_TopDbgCtl	rmw bits 24:26, 54:56, 61:63 = ‘100 000 000’	
x‘51 1400’	IF1TRC0	rmw 0, 1:4, 9:10 = ‘1 1111 00’	rmw bits 0, 9:10, 11:14 = ‘1 11 1111’
x‘51 1000’	IF0TRC0	rmw bit 0 = ‘1’	

Address Offset	Register Short Name	Disable Word 0	Disable Word 2
x‘50 8600‘	IOC_DTB_Cfg0	rmw bits 16:19 = ‘0000’	
x‘51 1C70’	IOC_TopDbgCtl	rmw bits 24:26 = ‘000’	

Bit	Type	Description
8	E	Received MMIO read targeted to IOIF1.
9	E	Received MMIO write targeted to IOIF1.
10	E	Received MMIO read targeted to IOIF0.
11	E	Received MMIO write targeted to IOIF0.
12	S	Sent command to IOIF0.
13	S	Sent command to IOIF1.

C.8.6 I/O Controller Word 2 - Group 2 (NClk/2)

Address Offset	Register Short Name	Enable Word 0	Enable Word 2
x‘50 8600’	IOC_DTB_Cfg0	rmw bits 0, 24:27 = ‘1 1111’	
x‘51 1C70‘	IOC_TopDbgCtl	rmw bits 33:35 = ‘000’	
x‘51 1400’	IF1TRC0	rmw bits 0, 1:4, 9:10 = ‘1 1111 11’	rmw bits 0, 9:10, 11:14 = ‘1 00 1111’
x‘51 1000’	IF0TRC0	rmw bit 0 = ‘1’	

Address Offset	Register Short Name	Disable Word 0	Disable Word 2
x‘50 8600‘	IOC_DTB_Cfg0	rmw bits 24:27 = ‘0000’	

Bit	Type	Description
5	B	IOIF0 Dependency Matrix 3 is occupied by a dependent command (see **Note**).
6	B	IOIF0 Dependency Matrix 4 is occupied by a dependent command (see **Note**).
7	B	IOIF0 Dependency Matrix 5 is occupied by a dependent command (see **Note**).
10	E	Received read request from IOIF0.
11	E	Received write request from IOIF0.
14	E	Received interrupt from the IOIF0.

Note: The dependency matrix (DM) entries are numbered 0-15. The IOC logic generally uses the lowest-numbered available slot for new incoming requests. Therefore, the lower numbered DM entries tend to be used more often than the higher-numbered entries.

C.8.7 I/O Controller - Group 3 (NClk/2)

Address Offset	Register Short Name	Enable Word 0	Enable Word 2
x‘50 8600’	IOC_DTB_Cfg0	rmw bits 0, 40:43 = ‘1 1111’	rmw bits 0, 32:35 = ‘1 1111’
x‘50 8618‘	IOC_DTB_Cfg3	rmw bit 47 = ‘1’	
x‘51 1C00’	IOC_IOCmd_Cfg	rmw bit 17 = ‘1’	
x‘51 1400’	IF1TRC0	rmw bits 0, 9:10 = ‘1 00’	
x‘51 1000’	IF0TRC0	rmw bits 0, 9:10 = ‘1 00’	

Address Offset	Register Short Name	Disable Word 0	Disable Word 2
x‘50 8600‘	IOC_DTB_Cfg0	rmw bits 32:35, 40:43 = ‘0000 0000’	
x‘50 8618‘	IOC_DTB_Cfg3	rmw bit 47 = ‘0’	

Bit	Type	Description
16	S	IOIF0 request for token for even banks (0-14); valid only if resource allocation is enabled.
17	S	IOIF0 request for token for odd banks (1-15); valid only if resource allocation is enabled.
18	S	IOIF0 request for token 1, 3, 5, or 7; valid only if resource allocation is enabled.
19	S	IOIF0 request for token 9, 11, 13, or 15; valid only if resource allocation is enabled.
24	S	IOIF0 request for token 16; valid only if resource allocation is enabled.
25	S	IOIF0 request for token 17; valid only if resource allocation is enabled.
26	S	IOIF0 request for token 18; valid only if resource allocation is enabled.
27	S	IOIF0 request for token 19; valid only if resource allocation is enabled.

C.8.8 I/O Controller Word 0 - Group 4 (NClk/2)

Address Offset	Register Short Name	Enable Word 0	Enable Word 2
x‘50 8600’	IOC_DTB_Cfg0	rmw bits 0, 48:51 = ‘1 1111’	
x‘51 1400‘	IF1TRC0	rmw bit 0 = ‘1’	
x‘51 1000’	IF0TRC0	rmw bits 0, 9:10, 11:14 = ‘1 00 1111’	rmw bits 0, 1:4, 9:10 = ‘1 1111 11’

Address Offset	Register Short Name	Disable Word 0	Disable Word 2
x‘50 8600‘	IOC_DTB_Cfg0	rmw bits 48:51 = ‘0000’	

Bit	Type	Description
0	E	I/O page table cache hit (for commands from IOIF).
1	E	I/O page table cache miss (for commands from IOIF).
3	E	I/O segment table cache hit.
4	E	I/O segment table cache miss.
24	E	Interrupt received from any SPU (reflected command when IIC decides to send an ACK response).
25	E	Internal interrupt controller (IIC) generated interrupt to PPU thread 0.
26	E	IIC generated interrupt to PPU thread 1.
27	E	Received external interrupt (using MMIO) from PPU to PPU thread 0.
28	E	Received external interrupt (using MMIO) from PPU to PPU thread 1.

Glossary

ABI	Application Binary Interface. The standard that a program follows to ensure that code generated by different compilers (and perhaps linking with various, third-party libraries) will run correctly on the CBE processor. The ABI defines data types, register usage, calling conventions, and object formats.
AC	(1) Address Compare bit. (2) Address Concentrator.
aligned reference	A memory reference to data that resides at an address that is an integral multiple of the data size.
AltiVec	The Freescale SIMD instruction set that is identical to IBM's Vector/SIMD Multimedia Extension SIMD instruction set. AltiVec is a trademark of Freescale.
ALU	Arithmetic Logic Unit.
AOS	Array Of Structures. A method of organizing related data values. Also called the vector-across, or vec-across, form. Compare *SOA*
API	Application Program Interface.
application	(1) A program designed to perform a specific task or group of tasks. (2) A user program, as opposed to a system or supervisor program; also called a *problem state* program in the *PowerPC Architecture*.
ARPN	Abbreviated Real Page Number.
asynchronous	Not coordinated in time with the execution of instructions in an instruction pipeline.
ATO	Atomic Unit. Part of an SPE's MFC. It is used to synchronize with other processor units.
atomic operation	A set of operations, such as read-write, that are performed as an uninterrupted unit. An atomic operation is a form of a mutual-exclusion (mutex) lock that can implement a semaphore.
atomic access	A bus access that attempts to be part of a atomic operation.
available to software	A region of memory that is not identified as allocated in *Figure 5-1 CBE Memory Map* on page 120, and that are known to be populated with memory chips or external MMIO registers, are available to software for any purpose.
AVPN	Abbreviated Virtual Page Number.
b	Bit.
B	(1) Byte. (2) Blocking.

barrier A command that ensures program-ordering of memory accesses of all preceding, non-immediate MFC commands with respect to all commands following the barrier command within the same MFC command queue. Compare *fence*.

basic block A sequence of instructions that has no branching or other transfer of control.

basic-block aggregation A method of extracting SIMD parallelism within a basic block by packing isomorphic computation on adjacent memory accesses into vector operations.

BClk Bus Interface Controller (BIC) Core Clock.

beat Data sent in one L2 clock cycle, which is two CBE Core-Clock cycles. Also called *data beat*.

BEI Cell Broadband Engine Interface Unit.

BHT Branch History Table.

BIC Bus Interface Controller. Part of the Cell Broadband Engine Interface (BEI) Unit to I/O.

BIF (1) Cell Broadband Engine Interface protocol. The EIB's internal communication protocol. It supports coherent interconnection for to other CBE processors and BIF-compliant I/O devices, such as memory subsystems, switches, and bridge chips. The protocol is software-selectable only at Power-On Reset (POR). See *IOIF*. (2) Branch indirect if false instruction.

big-endian An ordering of bytes and bits in which the lowest-address byte and lowest-numbered bit are the most-significant (high) byte and bit, respectively. The CBE processor supports only big-endian ordering; it does not support little-endian ordering.

BIU Bus Interface Unit. Part of the PPE's interface to the EIB.

block (1) A cache line, which is 128 bytes in size. (2) The inability for an instruction in a pipeline to proceed. Blocking occurs at the instruction-dispatch stage and, in the PPE, stops only one of the two threads. Compare *stall* and *bubble*.

blocking A property of certain SPE channels when its count is zero. Blocking channels cause the SPE to stall when the SPE reads or writes a channel with a count of zero. See *channel count* for more details.

boundedly undefined Instruction-execution results that could have occurred by executing an arbitrary sequence of instructions, starting from a given machine state.

branch-and-link A PPE branch instruction that writes the current instruction address plus 4 into the Link Register (LR).

branch hint A type of branch instruction that provides a hint of the address of the branch instruction and the address of the target instruction. Hints are coded by the programmer or inserted by the compiler. The branch is assumed taken to the target. Hints are used in place of branch prediction in the SPU.

BRU Branch Unit.

bubble An unused stage in a pipeline during a cycle. Compare *stall* and *block*.

built-ins The subset of *generic intrinsics* that map to assembly-language instruction sequences. A built-in cannot be expressed as a sequence of generic intrinsics, although it can be expressed as a sequence of *specific intrinsics*.

bypassing the cache Sending a cache line obtained from the L2 cache or main storage directly to the instruction pipeline, instead of first writing it into the L1 cache and then rereading it.

C (1) The C programming Language. (2) The Change bit in a PPE page-table entry.

cache High-speed memory close to a processor. A cache usually contains recently-accessed data or instructions, but certain cache-control instructions can lock, evict, or otherwise modify the caching of data or instructions.

cache block 128 bytes. Same as *cache line*.

cacheable An access, such as a load, store, or instruction fetch, in which the caching-inhibited (I) bit in the PTE is cleared to ‘0’.

caching-inhibited A memory update policy in which caches are bypassed, and the load or store is performed to or from main storage. Only the storage location specified by the instruction (rather than a full cache line) is accessed at a caching-inhibited location. Stores to caching inhibited pages must update the memory hierarchy to a level that is visible to all processors and devices in the system. The operating system typically implements this policy, for example, for I/O devices. The “I” bit in the “WIMG” bits.

cache line 128 bytes. Same as *cache block*.

cache touch A compiler-directed method, initiated by the **dcbt** and **dcbtst** instructions, of preloading data into the cache so that the subsequent accesses hit in the cache.

callee A program or routine that is called by another program or routine (the *caller*).

caller A program or routine that calls another program or routine (the *callee*).

cast out The process of writing modified data from a cache to the next-lower-level cache (for example, L1 to L2) or to main storage, and marking the cache line invalid.

CBE processor The Cell Broadband Engine chip.

CBEA *Cell Broadband Engine Architecture*. The CBE processor is one implementation of the *Cell Broadband Engine Architecture*.

CBE core clock NClk.

CEC Central Electronics Complex.

CBE task A task running on the PPE and SPE. Each such task has one or more main threads and some number of SPE threads, and all the main threads within the task share the task's resources, including access to the SPE threads.

CESOF An extension of PPE-ELF that allows PPE executable objects to contain SPE executables. The PPU and SPUs are supported by different compilers and different tools. CESOF allows a programmer to represent and resolve dependencies among PPU and SPU programs, which do not share a symbol space.

channel A 32-bit, unidirectional communication interface between an SPE and the system, including main storage, the PPE, and other SPEs.

channel count The value returned by the read-channel-count instruction (**rchcnt**). Reading the channel count of an implemented and blocking (queue-connected) channel returns the channel's available capacity. Reading the channel count of an implemented but nonblocking channel always returns a 1. Reading the channel count of an unimplemented (reserved) channel always returns a 0.

CIDR Class ID Register.

CIU Core Interface Unit.

CL A class-ID parameter in an MFC command.

class ID See `RClassID` and `TClassID`.

coherence The correct ordering of stores to a memory address, and the enforcement of any required cache write-backs during accesses to that memory address. Cache coherence is implemented by a hardware snoop (or inquire) method, which compares the memory addresses of a load request with all cached copies of the data at that address. If a cache contains a modified copy of the requested data, the modified data is written back to memory before the pending load request is serviced.

The primary objective of a coherent memory system is to provide the same image of memory to all devices in the system.

coherence domain All processor elements and all interfaces to main storage.

coherency block The block size used in managing memory coherence. In the CBE processor, it is 128 bytes.

coherence granularity See *coherency block*.

combined snoop response The snoop responses from all snoopers are combined together to form the combined snoop response that is sent to all devices in the system.

committed The point in the execution of an instruction at which all older instructions in the pipeline are past the flush point (the point at which they can be written back to main storage).

command A result of executing certain instructions. For example, an MFC DMA command can result from an SPU's execution of several write channel (**wrch**) instructions that define the parameters of a DMA transfer.

committed An instruction is said to be *committed* when the process of recording and writing back its result has begun and cannot be prevented by an exception.

common subexpression elimination The replacement of repeated, redundant expression evaluations with a single computation assigned to a temporary variable.

complete An instruction is said to *complete* when its result is both available to another instruction and can be written back to memory (retired), and past the point at which the instruction can cause an exception.

congruence class (1) A *set* (or row) of a set-associative cache, including a TLB cache. Compare *way*. (2) All cache entries indexed by a specified effective address. (3) A cache index.

constant folding A compiler's precalculation of constant expressions.

context (1) The environment (for example, privilege and relocation) of instruction execution that is controlled by system registers (such as the Machine State Register and Storage Description Register 1) and the relevant page table. (2) The process or task environment of instruction execution.

context switch A *process* or *task* switch.

context synchronizing The halting of instruction dispatch, clearing of the fetch buffer, and completion of all instructions currently in execution (that is, past the point at which they can cause an exception). The first instruction after a context-synchronizing event is fetched and executed in the context established by that instruction. Context synchronization occurs when certain instructions are executed (such as **isync** or **rfi***)* or when certain events occur (such as an exception). All context-synchronizing events are also execution synchronizing. Compare *execution synchronizing*.

control plane Refers to software or hardware that manages the operation of *data plane* software or hardware, by allocating resources, updating tables, handling errors, etc. See *data plane*.

core clock The CBE Core Clock (NClk).

count register	The PPE register that holds a loop count, which can be decremented during certain branch instructions, or which provides a branch target address.
CPI	Cycles Per Instruction.
CPL	Current Priority Level.
CR	Condition Register.
CSA	Context Save Area.
CSI	Context-Synchronizing Instruction.
CTR	Count Register.
cumulative ordering	The ordering of storage accesses performed by multiple sources.
cycle	Unless otherwise specified, one tick of the CBE Core Clock (NClk), which is the frequency at which the PPE and SPEs run.
DAR	Data Address Register.
data	(1) Any information in memory or on the processor's data bus, including instruction operands and instruction opcodes. (2) Instruction operands, as opposed to instruction opcodes.
data beat	See *beat*.
data hazard	A situation in which an instruction has a data dependence or a name dependence on a prior instruction, and they occur close enough together in the instruction sequence that the processor could generate a result inconsistent with execution in program order.
data plane	Refers to software or hardware that operates on a stream or other large body of data and is managed by *control plane* software or hardware. See *control plane*.
data stream	A sequence of contiguous data cache blocks (cache lines).
DCache	The PPE's L1 data cache.
dcbf	PPE data-cache-block flush instruction.
dcbst	PPE data-cache-block store instruction.
dcbt	PPE data-cache-block touch instruction.
dcbtst	PPE data-cache-block touch-for-store instruction.
dcbz	PPE data-cache-block set-to-zero instruction.
deadlock	A state in which two elements in a process are stalled, each waiting for the other to respond. Compare *livelock*.
DEC	Decrementer.

decrementer A register that counts down each time an event occurs. Each SPU contains dedicated 32-bit decrementers for scheduling or performance monitoring, by the program or by the SPU itself.

demand fetch A request by the L1 cache to the L2 cache for a cache line that caused a an L1 cache miss.

denormalized number A nonzero floating-point number whose exponent is the format's minimum, but represented as all zeros, and whose implicit leading significand bit is zero.

D-ERAT Data ERAT, or the data-cache effective-to-real-address translation table.

dependence A relationship between two instructions that requires them to execute in program order. Dependence is a property of a program.

DERR Data Error interrupt.

device memory Memory that has both the caching-inhibited and guarded attributes. This is typical of memory-mapped I/O devices.

DFQ Demand instruction request queue.

DIQ Denormalized Instruction Queue.

displacement An offset or index from a base address.

DLQ Demand data load request queue.

DMA Direct Memory Access. A technique for using a special-purpose controller to generate the source and destination addresses for a memory or I/O transfer.

DMAC Direct Memory Access Controller. A controller that performs DMA transfers.

DMA command A type of MFC command that transfers or controls the transfer of a memory location containing data or instructions. See *MFC command*.

DMA list A list of *list element*s that, together with an initiating DMA list command, specifies a sequence of DMA transfers between a single area of LS and discontinuous areas in main storage. Such lists are stored in an SPE's LS, and the sequence of transfers is initiated with a DMA list command such as **getl** or **putl**. DMA list commands can only be issued by programs running on an SPE, but the PPE or other devices can create and store the lists in an SPE's LS. DMA lists can be used to implement scatter-gather functions between main storage and the LS.

DMA list command A type of MFC command that initiates a sequence of DMA transfers specified by a DMA list stored in an SPE's LS. See *DMA list*.

DMA queue A queue for holding DMA-transfer commands. The MFC contains two DMA queues—a 16-entry SPU command queue and an 8-entry proxy command queue.

DMA transfer	Byte moves by a DMAC, without regard to the numerical significance of any byte.
dominant-shift reorganization	A reorganization of misaligned data in which the offset that is dominant (most often used) among a set of streams is chosen as the offset to shift to.
doubleword	Eight bytes.
DP	Double-precision.
DPFE	Data prefetch engine.
DR	Data relocate.
DSI	Data Storage Interrupt.
DSISR	Data Storage Interrupt Status Register.
DTS	Digital Thermal Sensor.
dual-issue	Issuing two instructions at once, under certain conditions. See *fetch group*.
dynamic branch prediction	Methods in which hardware records the resolution of branches and uses this information to predict the resolution of a branch when it is encountered again.
dynamic linking	Linking of a program in which library procedures are not incorporated into the load module, but are dynamically loaded from their library each time the program is loaded.
EA	Effective Address.
eager-shift reorganization	A reorganization of a misaligned load stream that shifts directly to the alignment of the store.
EAH	An effective address high parameter in an MFC command.
EAL	An effective address low parameter in an MFC command.
EAR	Effective-Address Reference. A software structure in the `.toe` section of the CESOF format.
ECC	Error-Correcting Code.
effective address	An address generated or used by a program to reference memory. A memory-management unit translates an effective address (EA) to a virtual address (VA), which it then translates to a real address (RA) that accesses real (physical) memory. The maximum size of the effective-address space is 2^{64} bytes.

EIB Element Interconnect Bus. The on-chip coherent bus that handles communication between the PPE, SPEs, memory, and I/O devices (or a second CBE processor). The EIB is organized as four unidirectional data rings (two clockwise and two counterclockwise).

eieio Enforce in-order execution of I/O.

ELF Executable and Linking Format. The standard object format for many UNIX® operating systems, including Linux. Originally defined by AT&T and placed in the public domain. Compilers generate ELF files. Linkers link .o files with ELF files in libraries. Systems run ELF files.

ELF header An *ELF* object-file header that is used to find a *section header* or a *program header*.

endian orientation A view of bits and bytes in which either the little end (least-significant or low end) or the big end (most-significant or high end) is assigned the lowest number or address. Thus, there are two types of endian orientation—*little-endian* and *big-endian*. Endian orientation applies to bits, in the context of register-value interpretation, and to bytes, in the context of memory accesses. The CBE processor supports only big-endian ordering; it does not support little-endian ordering.

ERAT Effective-to-Real Address Translation, or a buffer or table that contains such translations, or a table entry that contains such a translation.

ESID Effective segment ID

even pipeline Part of an SPE's dual-issue execution pipeline. Also referred to as pipeline 0.

execution synchronizing The completion of previously issued instructions—to a point where they can no longer cause an exception, but not necessarily in the context (privilege, protection, and address translation) under which they were issued—before continuing with program execution. Execution-synchronizing does not empty (flush) the instruction prefetch queue. The **sync** instruction is execution-synchronizing. Compare *context synchronizing*.

exception An error, unusual condition, or external signal that may alter a status bit and will cause a corresponding interrupt, if the interrupt is enabled. See *interrupt*, *imprecise interrupt*, and *precise interrupt*.

external access From an SPE's viewpoint, a read or write access by any DMA transfer (by the local MFC or the MFC associated with another SPE) or any processor (including the PPE or another SPE) or other device—other than the SPE being referred to.

fence A modifier to a DMA command that ensures that this command is ordered with respect to all preceding commands in the DMA command queue within the same tag group. Compare *barrier*.

fetch To read instructions (but not data) from storage.

fetch group A doubleword-aligned instruction pair. Dual-issue occurs when a fetch group has two instructions that are ready to issue, and when the first instruction can be issued on the even pipeline and the second instruction can be issued on the odd pipeline.

FIFO First In First Out. Refers to one way elements in a queue are processed. It is analogous to “people standing in line.”

FIR Fault Isolation Register.

fixed-point value An integer.

FlexIO Rambus FlexIO bus.The physical-link I/O signals on the BIF and IOIF interfaces. See *IOIF*.

FLIH First-Level Interrupt Handler.

flush (1) Write the contents of a cache line back to main storage, and invalidate the cache line. (2) Invalidate a cache line or an instruction pipeline.

flush point The execution-pipeline point at which all instructions older than the flush point can be written back to main storage.

Flush transaction A transaction caused by a **dcbf** that hits on a memory-coherent cache line and is marked shared or invalid. It is sent to other snoopers who might have a copy of the line.

FP Floating Point.

FPR Floating-Point Register.

FPSCR Floating-Point Status and Control Register.

FPU Floating-Point Unit.

fres Floating reciprocal estimate single A form instruction.

FXU Fixed-Point Unit. In the PPE, the fixed-point integer unit.

G The guarded bit in a page table entry which controls the processor's accesses to cache and main storage. It is part of the “WIMG” bits.

GCD The Greatest Common Divisor of two integers. It is the largest integer that divides them both. This is usually denoted by `GCD(a,b)`.

general-purpose registers The registers used for integer and string operations.

generic intrinsics Intrinsics that map to one of several assembly-language instructions or instruction sequences, depending on the type of operands. (See *intrinsic*.) All generic SPU intrinsics are prefaced by the string, `spu_`. For example, the generic intrinsic that implements the `stop` assembly instruction is named `spu_stop`.

get_ps An operating system call that returns a base address for the *problem state* area.

getllar Get lock line and reserve MFC command.

GPR General-Purpose Register.

guarded Regions of memory in which prefetching and other speculative storage operations are not permitted. A data access to a guarded location is performed only if either (a) the access is caused by an instruction that is required for sequential execution, or (b) the access is a load and the storage location is already in a cache. Such regions in memory are marked by setting the G bit in the relevant page-table entry. The operating system typically implements guarding, for example, on I/O devices.

H Hash Function Identifier.

harvest To reset (clear) the state of an SPE.

hashed page table A variable-sized, chained data structure that maps virtual page numbers (VPNs) and real page numbers (RPNs). A hash function is applied to a VPN to yield an entry into the page table. Each entry in a hashed page table is associated with a page in real memory. Also called *inverted page table*.

hazard A situation in which the overlapped or out-of-order execution of a pair of instructions could generate a result inconsistent with execution of the instructions in program order. A hazard is a property of a program.

HDEC Hypervisor Decrementer.

hcall Hypervisor call, which is an **sc** instruction with the LEV bit set to ‘1’.

HID Hardware-Implementation Dependent.

high The most-significant bit or byte numbers in a field, register or memory. High bits and bytes are the lowest-numbered bits or bytes in a data structure. For example, bit 0 is the high bit of a bit field or register.

hint stall The holding of a branch instruction in or before the stage of the pipeline in which triggering occurs. The branch is held there until an associated hint-for branch trigger is loaded.

hoist To move an instruction to an earlier point in the program-execution order.

HRMOR Hypervisor Real-Mode Offset Register.

HTAB Hashed page table.

hypervisor A control (or virtualization) layer between hardware and the operating system. It allocates resources, reserves resources, and protects resources among (for example) sets of SPEs that may be running under different operating systems.

The CBE processor has three operating modes: user, supervisor and hypervisor. The hypervisor performs a meta-supervisor role that allows multiple independent supervisors' software to run on the same hardware platform.

For example, the hypervisor allows both a real-time operating system and a traditional operating system to run on a single PPE. The PPE can then operate a subset of the SPEs in the CBE processor with the real-time operating system, while the other SPEs run under the traditional operating system.

hypervisor call An **sc** instruction with the LEV bit set to '1'.

I The caching-inhibited bit in a page table entry which controls the processor's accesses to cache and main storage. It is part of the "WIMG" bits.

IBuf Instruction Buffer.

ICache The PPE's L1 instruction cache.

icbi PPE instruction-cache-block invalidate instruction.

ID Instruction Dispatch.

IEEE 754 The IEEE 754 floating-point standard. A standard written by the Institute of Electrical and Electronics Engineers that defines operations and representations of binary floating-point arithmetic.

I-ERAT Instruction ERAT, or instruction-cache effective-to-real-address translation table.

IFAR Instruction Fetch Address Register.

IGP Interrupt Generation Port. One of two registers, `IIC_IGP0` and `IIC_IGP1` (one for each PPE thread), to which software on the PPE or an SPU can write an Interprocessor Interrupt (IPI).

IIC Internal Interrupt Controller.

ILP Instruction-level parallelism.

ILP32 32-bit Integers, Longs, and Pointers. The SPEs are always ILP32, with DMA support for 32- and 64-bit effective-address pointers

immediate operand An operand included in an instruction. Also called *immediate constant* or *immediate value*.

imprecise interrupt A synchronous exception that does not adhere to the precise exception model. Imprecise interrupts occur one or more instructions after the execution of the instruction causing the interrupt. They are not restartable. In the CBE processor, single-precision floating-point operations generate imprecise exceptions. See *precise interrupt*.

inbound access An access initiated externally from an I/O device (also called I/O unit) to the CBE processor. Also called an *I/O access*. Compare *outbound access*.

index An offset from a base address.

indirect An access is said to be indirect if a register holds its target. For example, an indirect branch is one whose target is specified in a register.

inline expansion An optimization in which the reference to a procedure is replaced with the code of the procedure itself to eliminate calling overhead.

in-order In program order. The PPE and SPEs execute instructions in-order; i.e., they do not rearrange them (out-of-order).

instruction-sequencing error An error that occurs when instructions are executed in a different order other than expected. The SPU expects instructions to be executed sequentially unless modified by hint trigger.

instruction runout A situation in which there are no instructions remaining in the SPU instruction pipeline, and the SPU becomes idle.

interrupt A change in machine state in response to an exception. See *exception*, *imprecise interrupt*, and *precise interrupt*.

interrupt packet Used to signal an interrupt, typically to a processor or to another interruptible device.

interrupt vector The starting address of an interrupt handler or service routine.

intervention The peer-to-peer movement of data between two caches, without involving main storage.

intrinsic An in-line set of assembly-language instructions in the form of function call that is built into a compiler. Intrinsics make the underlying ISA accessible from a high-level programming language.

inverted page table See *hashed page table*.

I/O access (1) An I/O operation that is initiated internally or externally to the CBE processor. (2) As used in *Section 7.7 I/O Storage Model* on page 187, an I/O operation that is initiated externally to the CBE processor.

I/O address As used in *Section 7.7 I/O Storage Model* on page 187, an address passed from an I/O device to the CBE processor when the I/O device attempts to perform an access to the CBE processor's real-address space.

IOC	I/O Interface Controller.
I/O device	Input/output device. From software's viewpoint, I/O devices exist as memory-mapped registers that are accessed in main-storage space by load/store instructions. The operating system typically configures access to I/O devices as caching-inhibited and guarded.
IOID	I/O Identifier. It is described in the IOPT and identifies an I/O unit.
IOIF	(1) One of two I/O interfaces supported by the EIB. Also called a FlexIO interface. (2) A non-coherent communication protocol used by I/O devices attached to an IOIF, which differs from the coherent BIF protocol. The IOIF protocol can be used on either IOIF0 or IOIF1; the BIF protocol can be used only on IOIF0. The protocol is software-selectable only at Power-On Reset (POR).
IOIF0, IOIF1	One of two I/O (IOIF) interfaces, also called a FlexIO interfaces.
IOIF device	A device that is connected to CBE processor's IOIF port directly.
I/O operation	A storage operation that crosses a Cell Broadband Engine coherence-domain boundary.
IOIF protocol	The EIB's noncoherent protocol for interconnection to I/O devices. See *BIF*.
IOPT	I/O Page Table.
IOST	I/O Segment Table.
I/O unit	One or more physical I/O device(s), I/O bridge(s), or other functional unit(s) attached to an IOIF, in which one value of the IOID described in the IOPT. A functional unit that can initiate I/O accesses.
IPC	Instructions Per Cycle.
IPFQ	Instruction prefetch request queue.
IPI	Interprocessor Interrupt. In the CBE processor, a software interrupt written by a PPE thread or an SPE to one of the Interrupt Generation Port (IGP) registers.
IR	Instruction Relocate.
IPP	Interrupt Pending Port.
IS	(1) Instruction issue. (2) Invalidation Selector.
ISEG	Instruction Segment Exception.
ISI	Instruction Storage Interrupt.
ISRC	Interrupt Source.
IU	Instruction Unit.

K	2^{10} (as in KB for 1,024 bytes).
KB	Kilobyte.
key	A value, ‘0’ or ‘1’, used in conjunction with the page protection (PP) bits in page-table entries to determine access rights to locations in main storage. The value of the key is equal to the value of the `Ks` bit or value of the `Kp` bit in an SLB entry, depending on the `PR` bit in the machine state register (`MSR`).
Kp	The problem-state (user-mode) storage key bit in SLB entries. In address translations, the `Kp` bit is used in conjunction with the `PR`, `PP`, `N`, and `Ks` bits to determine access privilege.
Ks	The supervisor-state storage key bit in SLB entries. In address translations, the `Ks` bit is used in conjunction with the `PR`, `PP`, `N`, and `Ks` bits to determine access privilege.
L	Large Page Indicator.
L1	Level-1 cache memory. The closest cache to a processor, measured in access time.
L2	Level-2 cache memory. The second-closest cache to a processor, measured in access time. An L2 cache is typically larger than an L1 cache.
LAR	Load and reserve.
LARX-reserve transaction	An address-only transaction that sets the reservation for every cache level below the level serviced by a read atomic operation.
latency	(1) The number of cycles between when an instruction begins execution to when the result is available. (2) The issue-to-issue latency for a dependent instruction. (3) The time between when a function is called and when it returns.
lazy-shift reorganization	A reorganization of misaligned data that delays eager-shift reorganization as long as possible.
ld	PPE load double word instruction.
ldarx	PPE load-double-word-and-reserve indexed instruction.
LEAL	List-element Effective-Address Low. A parameter in a DMA-list MFC command.
least-recently used	An algorithm for replacing cache entries to identifies or approximates the oldest entry so that it can be *cast out* (evicted) to make room for a new entry.
least-significant	The highest-numbered bits or bytes (big-endian) in a data structure.
length conversion	A conversion between data of different lengths.

link register The PPE register used to provide a branch-target address and to hold the return address for certain branch instructions.

linker A program that resolves cross-references between separately compiled or assembled object modules and then assigns final addresses to create a single relocatable load module. If a single object module is linked, the linker simply makes it relocatable. Also called a *link editor*.

list element Also called *transfer element*. See *DMA list*.

livelock A state in which processors interact in a way such that no processor makes forward progress, such as in an endless loop of program execution. Compare *deadlock*. In a livelock, processing continues to take place. In a deadlock, no processing continues.

lmw PPE load multiple word instruction.

lnop A NOP in an SPU's odd pipeline. It can be inserted in code to align for dual issue of subsequent instructions.

load combining The combining of multiple load accesses to storage into a single load access.

loader A program that reads the load module into main storage, performing all necessary dynamic linking so that the module can execute.

local storage An abstraction of *local store*, as specified by the *Cell Broadband Engine Architecture*.

local store An implementation of an SPE's *local storage*. The 256-KB on-chip memory associated with each SPE. It holds both instructions and data. Abbreviated as LS.

lock (1) To use a Replacement Management Table (RMT) entry to prevent L2-cache or TLB entries from being *cast out*. (2) To obtain exclusive access to a main-storage location using the **lwarx** and **stwcx.** instructions. (3) See *pinned*.

logical partition A virtual machine managed by a hypervisor.

logical PPE One of the two PPE threads.

loop distribution The splitting off of those parts of a loop that cannot be SIMDized

loop interchange The swapping of an inner (nested) loop with an outer loop. This is done when the inner loop appears to be the candidate for SIMDization.

loop-level aggregation A method of extracting SIMD parallelism across loop iterations. Computations on stride-one accesses across loop iterations can be combined into vector operations.

loop unrolling A programming optimization that increases the step of a loop, and duplicates the expressions within a loop to reflect the increase in the step. This can improve instruction scheduling and memory access time.

loop versioning	A technique for hoisting an individual array-index exception-check outside a loop by providing two copies of the loop: the safe loop and the unsafe (original) loop.
looped-back operation	A command that is received by the Cell Broadband Engine from an IOIF device and routed back out the same IOIF.
low	The least-significant bit or byte numbers in a field, register or memory. Low bits and bytes are the highest-numbered bits or bytes (big-endian) in a data structure. For example, bit 63 is the low bit of a doubleword field or register.
LP	Large Page Selector.
LP64	64-bit Longs and Pointers. 32-bit Integers.
LPAR	Logical Partition.
LPCR	Logical Partition Control Register.
LPES	Logical Partitioning Environment Selector. A field in the LPCR register.
LPID	Logical Partition ID.
LR	Link Register.
LRU	Least Recently Used. See *least-recently used*.
LS	See *local store*.
LSA	Local Store Address.
LSCSA	Local Store Context Save Area,
LSU	Load and Store Unit.
lswi	PPE load string word immediate instruction.
lswx	PPE load string word indexed instruction.
LSZ	List Size. A parameter in an MFC command.
LTS	List-element Transfer Size. A parameter in an MFC command.
lwarx	PPE load-word-and-reserve indexed instruction.
m	The number of bits in a real address. For the CBE processor, this value is 42.
M	(1) The memory-coherence bit in a page table entry which controls the processor's accesses to cache and main storage. It is part of the “WIMG” bits. (2) 2^{20}, as in MB for 1,048,576 bytes.

mailbox A queue in an SPE's MFC for exchanging 32-bit messages between the SPE and the PPE or other devices. Two mailboxes (the SPU Write Outbound Mailbox and SPU Write Outbound Interrupt Mailbox) are provided for sending messages from the SPE. One mailbox (the SPU Read Inbound Mailbox) is provided for sending messages to the SPE.

main memory See *main storage*.

main storage (1) The effective-address (EA) space. It consists physically of physical memory (whatever is external to the memory-interface controller), SPU LSs, memory-mapped registers and arrays, memory-mapped I/O devices (all I/O is memory-mapped), and pages of virtual memory that reside on disk. It does not include caches or execution-unit register files. (2) The level of storage hierarchy in which all storage state is visible to all processors and mechanisms in the system. See *storage*

mangle To add an extra prefix or suffix to a global symbol that is produced by a compiler. Global symbols produced by a compiler are said to be mangled if a symbol has any extra prefix or suffix added by the tool chain.

MB Megabyte.

MC Microcode.

memory-mapped Mapped into the Cell Broadband Engine's addressable-memory space. Registers, SPE local stores (LSs), I/O devices, and other readable or writable storage can be memory-mapped. Privileged software does the mapping.

memory stream A sequence of contiguous memory locations that are accessed by a memory reference throughout the lifetime of a loop.

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

MFC Memory Flow Controller. It is part of an SPE and provides two main functions: moves data via DMA between the SPE's *local store* (LS) and main storage, and synchronizes the SPU with the rest of the processing units in the system.

MFC command A command issued to an SPE's MFC. The command and any related parameters are written with a series of instructions executed either on the MFC's associated SPU or on the PPE or other device. MFC commands provide the main mechanism by which software running on an SPU accesses main storage and maintains synchronization with other devices in the system. DMA commands are a primary type of MFC command.

MFC Pause MFC Pause state. One of six power-management states.

MFC proxy commands MFC commands issued using the MMIO interface.

mfceieio	MFC enforce in-order execution of I/O command.
mfcsync	MFC synchronize command.
mfspr	PPE move from special-purpose register instruction.
MIC	Memory Interface Controller. The CBE processor's MIC supports two memory channels.
MiClk	MIC Core Clock.
microarchitecture	A microprocessor's hardware architecture.
miss penalty	The time required to fill a cache line after a cache miss.
mixed-mode SIMDization	The devirtualization of virtual vectors and their replacement by a single SIMD instruction, a sequence of SIMD instructions, a library call, or a sequence of scalar operations.
MMIO	Memory-Mapped Input/Output. The documentation for the CBE processor defines an "MMIO register" as any internal or external register that is accessed through the main-storage space with load and store instructions, whether or not the register is associated with an I/O device. See *memory-mapped*.
MMU	Memory Management Unit. A functional unit that translates between effective addresses (EAs) used by programs and real addresses (RAs) used by physical memory. The MMU also provides protection mechanisms and other functions.
M:N thread model	A programming model in which M user threads are mapped to N kernel threads (or virtual processors).
MOD	Modified snoop response code.
MODINTV	Modified-intervention snoop response code.
most-significant	The lowest-numbered bits or bytes (big-endian) in a data structure.
MPI	Message Passing Interface. A standard for high-performance communication on massively parallel architectures and clustered distributed-memory systems.
MSb	Most-Significant bit.
MSR	Machine State Register.
MT	Multithreading. See *multithreading*.
mtmsr	PPE move to machine state register instruction.
mtspr	PPE move to special-purpose register instruction.

multithreading Simultaneous execution of more than one program thread. It is implemented by sharing one software process and set of execution resources but duplicating the architectural state (registers, program counter, flags, etc.) of each thread.

mutex lock Mutual-exclusion lock. An atomic operation used to implement a semaphore.

n The number of bits in a virtual address. For the CBE processor, this value is 65.

N (1) No Execute bit in SLB entries. (2) Intervention bit. (3) Non-Blocking. (4) Normal run thermal-management state.

NaN Not-a-Number. A special string of bits encoded according to the IEEE 754 Floating-Point Standard. A NaN is the proper result for certain arithmetic operations; for example, 0/0 = NaN. There are two types of NaNs, quiet NaNs and signaling NaNs. Only the signaling NaN raises a floating-point exception when it is generated.

NClk CBE Core Clock. The clock for the PPU and SPU. It is the highest-frequency CBE clock.

NCU Non-Cacheable Unit.

nop No operation. Also called *no-op*.

no-op See *nop*.

NPC An SPE's Next Program Counter register.

NUMA Non-uniform memory access.

object module The output file of a compiler or other language translator. It includes the machine language translation and other information for symbolic binding and relocation.

odd pipeline Part of an SPE's dual-issue execution pipeline. Also referred to as pipeline 1.

offset An index that is added to a base address.

OGSA Open Grid Services Architecture. A standard that enables communication across heterogeneous, geographically dispersed environments.

OpenMP Open Multiprocessing. An API that supports multiplatform, shared-memory parallel programming.

ordered Said of an exception that only generates an interrupt in a prioritized order, with respect to the state of the interrupt-processing mechanism.

OS Operating System.

out of order Not in program order.

outbound access An access initiated internally from the CBE processor to an I/O device (also called I/O unit). Compare *inbound access*.

overlay SPU code that is dynamically loaded and executed by a running SPU program.

p Page size. A power-of-2 variable representing the size of a page. Three concurrent page sizes can be used: 4 KB (p = 12) and any two of the following large page sizes: 64 KB (p = 16), 1 MB (p = 20), 16 MB (p = 24).

page A unit of main-memory storage. Each page can have independent protection and control attributes, and Change and Reference status bits can be independently recorded. See *page table*.

page fault A restartable interrupt that causes the loading of a page from disk to memory.

page-history recording Same as *storage-access recording*.

page table A table that maps pages of virtual addresses (VAs) to real addresses (RA) and contains related protection parameters and other information about memory locations. See *hashed page table*.

page-table entry group A 64-byte data structure in the hashed page table that contains eight page-table entries.

parity A means of checking data reliability in which data bits are concatenated with a parity bit whose value makes the total number of ‘1’ bits even or odd.

path length The number of instructions in an instruction sequence.

PC (1) Program Counter. The PPE maintains separate program counters for each thread. (2) Personal Computer.

performance simulation Simulation by the IBM Full System Simulator for the Cell Broadband Engine in which both the functional behavior of operations and the time required to perform the operations is simulated. Also called *cycle-accurate simulation*.

pervasive logic Logic that provides power management, thermal management, clock control, software-performance monitoring, trace analysis, and so forth.

physical address (1) A *PowerPC Architecture* real address (RA). (2) An address on the CBE processor’s address-bus signals.

pinned A set of virtual-memory pages is said to be pinned or locked if the operating system has forced them resident in memory, so that they are not swapped out to disk. In addition to pages of memory being locked, the TLB entries associated with those pages can also be locked, thus preventing TLB misses.

pipelined hint mode An SPU pipelined mode in which a hint-for branch stall is negated and no further hint stall is generated for the duration of the pipelined hint mode. Pipelined hint mode lasts for 16 cycles after the last stallable hint is issued in pipelined hint mode.

pipelining A technique that breaks operations, such as instruction processing or bus transactions, into smaller stages so that a subsequent stage in the pipeline can begin before the previous stage has completed.

PIR Processor Identification Register.

plugin Fully resolved code that is dynamically loaded and executed by running an SPU program. Plugins facilitate code overlays.

PLG Physical Layer Group.

PLL Phase-Locked Loop.

POR Power-On-Reset for the CBE processor.

POSIX Portable Operating System Interface.

POSIX threads library The standard UNIX threads library, available at http://www-106.ibm.com/developerworks/linux/library/l-posix1.html. See *POSIX*.

PowerPC Of or relating to the *PowerPC Architecture* or the microprocessors that implement this architecture.

PowerPC Architecture A computer architecture that is based on the third generation of RISC processors. The *PowerPC Architecture* was developed jointly by Apple, Motorola, and IBM.

PowerPC 970 A 64-bit microprocessor from IBM in the PowerPC family. It supports both the PowerPC and Vector/SIMD Multimedia Extension instruction sets.

PP Page Protection bits.

PPE PowerPC Processor Element. The general-purpose processor in the CBE processor. It consists of the PPU and the PPSS.

PPE Pause (0) PPE Pause (0) state. One of six power-management states.

PPSS PowerPC Processor Storage Subsystem (L2 cache, NCU, CIU, BIU). Part of the PPE. It operates at half the frequency of the PPU.

PPU PowerPC Processor Unit. The part of the PPE that includes execution units, memory-management unit, and L1 cache.

pragma A compiler directive, inserted in source code by the programmer.

precise interrupt An interrupt in which the architecturally visible processor state (in the CBE processor, this is the PPE state) is known at the time of the exception that caused the interrupt, so that the pipeline can be stopped, instructions that preceded the faulting instruction can complete, and subsequent instructions can be flushed and redispatched after interrupt handling has completed. A precise interrupt is caused by an exception that was generated when the instruction was fetched or executed.

predicate A Boolean-logic term denoting a logical expression that determines the state of some variables. For example, a predicate can be an expression stating that variable A must have a value of 3.

predictive commoning A form of common subexpression elimination that exploits data reuse among consecutive loop iterations.

preferred scalar slot The preferred location for scalar values within a 128-bit register. The preferred location for 8-bit scalars is byte 3. The preferred location for 16-bit scalars is bytes 2 through 3. The preferred location for 32-bit scalars is bytes 0 through 3. The preferred location for 64-bit scalars is bytes 0 through 7. Compare *preferred slot*. See *Figure 3-6* on page 73.

preferred slot The left-most word (bytes 0, 1, 2, and 3) of a 128-bit register in an SPE. This is the SIMD word element in which scalar values are naturally maintained. Compare *preferred scalar slot*. See *Figure 3-6* on page 73.

prefetch To fetch instructions ahead of the processor's ability to dispatch them.

primitive A simple procedure, such as a test-and-set loop used for atomic lock acquisition.

Privilege 1 A designation given to SPE registers that are the most privileged of all SPE registers, used by the hypervisor to manage the SPE on behalf of a logical partition.

Privilege 2 A designation given to SPE registers that are the second-most privileged of all SPE registers, used by the operating system in a logical partition to manage the SPE within the partition.

Privilege 3 A designation given to SPE registers that are the least privileged of all SPE registers, used by Problem State (application) software, if direct access to the SPE from user space is supported by the operating system.

privileged state Also known as supervisor state. The permission level of operating system instructions. The instructions are described in the *PowerPC Operating Environment Architecture, Book III* and are required of software the accesses system-critical resources.

privileged software Software that can be executed only in supervisor or hypervisor state.

problem state The permission level of user instructions. The instructions are described in the *PowerPC Architecture* Books I and II and are required of software that implements application programs. Compare *supervisor state*.

program header An *ELF* object-file header specifies program segments in the file for use in program loading. Compare *section header*.

program order The order in which instructions occur in the program. Compare *sequential order*.

PRV Pervasive Logic. For details, see the *Cell Broadband Engine Hardware Initialization Guide*.

pseudo-LRU A cache-replacement algorithm that approximates, and performs faster than, a true *least-recently used* replacement algorithm.

PTE Page Table Entry. See *page table*.

PTEG Page Table Entry Group.

push A write, triggered by a snooped transaction, that copies a modified cache line to main storage. Also called *push out*.

putllc MFC put lock line conditional command.

putlluc MFC put lock line unconditional command.

QoS Quality of Service. It usually relates to a guarantee of minimum bandwidth for streaming applications.

quadword 16 bytes.

R Reference bit. It is set by hardware whenever the page is accessed and can be used by the operating system to determine recently used pages.

RA Real Address. See *real address*.

RAG Resource Allocation Group.

RAG U Unallocated Resource Allocation Group.

RAID Resource Allocation ID, or Resource Allocation Group ID. It is the ID of a Resource Allocation Group (RAG).

RAM Resource Allocation Management. A mechanism that allocates access to Resource Allocation Groups (RAGs). Examples are the allocation of access to memory banks or I/O interfaces.

range hit The satisfaction of all conditions specified in the Address Range Registers.

RAS Reliability, Availability, Serviceability.

RAW Read-After-Write dependency.

RC (1) Read-and-Claim. The L2 cache has six RC state machines that move data in and out of the L2 Cache in response to PPU and/or snoop requests. (2) The Reference (R) and Change (C) bits in a PTE.

Rc See *record bit*.

RC machine Read and Claim machine.

RC update An updates to the Reference (R) or Change (C) bits in a PTE.

rchcnt SPU read channel counter instruction.

RClassID Replacement Class ID.

rdch SPU read from channel instruction.

real address The address of a byte in real storage or on an I/O device. Real storage includes physical memory, the PPE's L1 and L2 caches, and the SPE's LSs. The maximum size of the real-address space is 2^{42} bytes.

real addressing mode A PPE addressing mode in which address translation is disabled (`MSR[IR]` = '0' for instructions, `MSR[DR]` = '0' for data). Accesses in real mode bypass storage protection checks. The effective address is used as the real address; on the PPE (but not on the SPEs), such a real address may be offset by an RMOR or HRMOR base address.

real mode Same as *real addressing mode*.

record bit Bit 31 (the Rc bit) in a PPE instruction opcode. When set to 1, the instruction updates the Condition Register (CR) according to the operation's result. Instructions that have the record bit set a dot (or period) suffix on their mnemonic.

recording The setting of bits in the Condition Register (CR) to reflect characteristics of an executed instruction's result. The recording is caused by instruction mnemonics that end in a period (.); such instructions cause the record (Rc) bit of the instruction format to be set to '1'. See *record bit*.

register spill A situation in which the instantaneous number of active variables exceeds the size of the register file.

register stream A sequence of contiguous registers that are produced by an operation over the lifetime of a loop.

relocation Virtual-address translation. Instruction and data relocation can be enabled independently using two bits (`IR` and `DR`) in the Machine State Register (`MSR`).

Replacement Management Table A software-controlled cache-replacement facility used to lock entries in a cache, thus preventing their replacement. The CBE processor provides an RMT for the PPE's TLB and L2 cache, and for each SPE's TLB.

reservation An exclusive right to access a main-storage location. Reservations are set and cleared with the **lwarx** and **stwcx** instructions, MFC atomic commands, and other instructions that access the reservation granule in which the reservation is set. Also called *lock*.

reservation granule The storage block size corresponding to the number of low-order bits ignored when a store to a real address is compared with a *reservation* at that address.

reserved Register locations marked as *reserved* and bit fields within registers marked as *reserved not implemented* operate in the same way: writes have no effect and reads return all ‘1’s. Reserved areas of the MMIO-register memory map that are not assigned to any functional unit should not be read from or written to: doing so will cause serious errors in software as follows. See *Reserved Regions of Memory and Registers* on page 30.

retire To write the results of a completed instruction back to main storage. Compare *complete*.

RI Recoverable Interrupt. A bit in the Machine State Register (`MSR`).

RMI Real-Mode Caching Inhibited.

RMLR Real Mode Limit Register.

RMSC Real-Mode Storage Control.

RMO Real-Mode Offset.

RMOR Real-Mode Offset Register.

RMR Range Mask Register.

RMT See *Replacement Management Table*.

RPN Real Page Number.

RSR Range Start Register.

runout See *instruction runout*.

runtime alignment Memory alignment cannot be known at compile-time and is instead done at runtime.

SBI Synergistic Bus Interface. The MFC’s interface to the EIB.

scalar A single data item, as opposed to a set of data items such as a vector or array.

scatter-gather A technique for operating on sparse data, using an index vector. A scatter-gather operation takes an *vector* and fetches data at an address added to that of the vector. A scatter operation stores data back to memory, using the same index vector.

scheduling A compiler optimization that reorders the instruction sequence subject to data- and control-flow restrictions so as to maximize use of a processor’s hardware.

SCN SPU Control Unit. A unit in SPU that handles branches and program control.

sdcrf SL1 data-cache-range flush.

sdcrt SL1 data-cache-range touch. Implemented as a nop on the CBE processor.

sdcrtst SL1 data-cache-range touch-for-store. Implemented as a nop on the CBE processor.

sdcrz SL1 data-cache-range set-to-zero.

SDR Storage Descriptor Register.

section header An *ELF* object-file header specifies sections in the file for use in linking. Compare *program header*.

segment A fixed 256-MB unit of address space that can hold code, data, or any mixture thereof. Segments are controlled by the Segment Lookaside Buffer (SLB), which maps EAs to VAs and provides protections.

semaphore A software flag used to indicate the status of, and lock the availability of, a shared resource. Semaphores can be implemented with atomic operations.

sequential order The order in which the compiler output of a program appears in main storage. Compare *program order*.

serialization A hardware-enforced alteration of the processor state or execution pipeline so as to match the sequential ordering of instructions in a program. For example, an instruction is said to be serializing if it (a) causes all preceding instructions, in program order, to complete before it begins execution, and (b) completes execution before any following instructions, in program order, begin execution. Compare *synchronization*. See also *program order, sequential order*, and *context synchronizing*.

set A row of a set-associative cache, including a TLB cache. Also called a *congruence class*. Compare *way*.

SFP SPU Floating-Point Unit. It handles single-precision and double-precision floating-point operations.

SFX SPU Even Fixed-Point Unit. It handles arithmetic, logical, and shift operations.

SFS SPU Odd Fixed-Point Unit. It handles shuffle operations.

short-loop aggregation The elimination of inner loops that have short trip counts. SIMDizable short loops can be collapsed into vector operations.

signal Information sent on a signal-notification channel. These channels are inbound (to an SPE) registers. They can be used by the PPE or other processor to send information to an SPE. Each SPE has two 32-bit signal-notification registers, each of which has a corresponding memory-mapped I/O (MMIO) register into which the signal-notification data is written by the sending processor. Unlike mailboxes, they can be configured for either one-to-one or many-to-one signalling.

These signals are unrelated to UNIX signals. See *channel*and *mailbox*.

signal notification See *signal*.

SIMD Single Instruction Multiple Data. Processing in which a single instruction operates on multiple data elements that make up a vector data-type. Also known as vector processing. This style of programming implements data-level parallelism.

SIMDize Transform scaler code to vector code.

single-ported Single-ported memory allows only one access at a time.

SL1 A first-level cache for DMA transfers between LS and main storage. The SPEs in the CBE processor do not implement SL1s.

SLB Segment Lookaside Buffer. It is used to map an effective address (EA) to a virtual address (VA).

SLBE SLB Entry.

slbia SLB invalidate all instruction.

slbie SLB invalidate entry instruction.

slbmfee SLB move from entry ESID X-form instruction.

slbmfev SLB move from entry VSID X-form instruction.

slbmte SLB move to entry X-form instruction.

SLIH Second-Level Interrupt Handler.

slow mode CBE Slow state. One of six power-management states.

SLS SPU Load and Store Unit. It handles loads, stores, and branch hints.

SMD Slow-Mode Divider. A clock-divider value in `PMSR1[BE_Slow]` that helps specify the maximum time-base frequency.

SMM Synergistic Memory Management Unit. It translates EAs to RAs for in an SPE.

SMP Symmetric Multiprocessor or Symmetric Multiprocessing. Two CBE processors can be connected together to form an SMP system.

sndsig Send signal command.

sndsigb Update signal-notification registers in an I/O device or another SPE with barrier command.

sndsigf Update signal notification registers in an I/O device or another SPE with fence command.

snoop To compare an address on a bus with a tag in a cache, in order to detect operations that violate memory coherency.

snoop push See *push*.

SNP Snoop.

SOA Structure of arrays. A method of organizing related data values. Also called parallel-array form. See *AOS*.

software cache Same as *software-managed cache*.

software-managed cache An SPE's local store (LS), which is filled from main memory using software-initiated DMA transfers. Although most processors reduce latency to memory by using hardware caches, an SPE uses its DMA-filled LS. The approach provides a high degree of control for real-time programming. However, the approach is advantageous only if the DMA transfer-size is sufficiently large and the DMA command is issued well before the data is needed, because the latency and instruction overhead associated with DMA transfers exceeds the latency of servicing a cache miss on the PPE.

software-managed memory Same as *software-managed cache*.

software pipelining A loop optimization in which the body of the loop in divided into a series of stages that are executed in parallel in a manner analogous to hardware pipelining.

southbridge A chip that interfaces a processor to I/O buses (except, usually, graphics).

SP (1) Single-Precision. (2) Stack Pointer.

specific intrinsic A type of C and C++ language extension that maps one-to-one with a single SPU assembly instruction. All SPU specific intrinsics are named by prefacing the SPU assembly instruction with `si_`.

SPE Synergistic Processing Element. It includes an SPU, an MFC and an LS. In this document, the term "SPE" refers generally to functionality of any part of the processor element, including the MFC, and the term "SPU" refers to the instruction set or the unit that executes the instruction set.

SPE SRI SPE State Retained and Isolated (SRI) state. One of six power-management states.

SPE thread A thread scheduled and run on an SPE. A program can have one or more SPE threads. Each thread has its own SPU *local store* (LS), register file, program counter, and MFC command queues.

spin lock	A synchronization primitive for shared-storage environments that executes a tight loop, attempting to acquire a lock at each iteration.
splat	To replicate, as when a single scalar value is replicated across all elements of a SIMD vector.
SPR	Special Purpose Register.
SPU	Synergistic Processor Unit. The part of an SPE that executes instructions from its local store (LS). In this document, the term “SPU” refers to the instruction set or the unit that executes the instruction set, and the term “SPE” refers generally to functionality of any part of the processor element, including the MFC.
SPU ISA	SPU Instruction Set Architecture. A SIMD instruction set executed in SPEs that is similar to the Vector/SIMD Multimedia Extension instruction set executed by the PPE.
SPU Pause	SPU Pause state. One of six power-management states.
SRR0	Save and Restore Registers 0.
SRR1	Save and Restore Registers 1.
SSC	SPU Channel and DMA Unit. It handles all input and output functions for an SPU.
SSE	Single SIMD Extensions. An Intel® instruction set.
stale	A value for a parameter is said to be stale if it is not the current value for that parameter.
stall	(1) The inability for an instruction in a pipeline to proceed. Possible causes of the stall include occupation of the next pipeline stage by another instruction, waiting for operands, or serialization. Stalls occur at the instruction-issue stage and, in the PPE, stop both threads. Compare *block* and *bubble*. (2) The number of cycles, after completing the execution of an instruction of a given type, before another instruction of the same type can be issued.
starvation	A condition in which a processing element is making forward progress, but at an extremely slow rate.
static branch prediction	A method in which software (for example, a compiler) gives a hint to the processor about the direction that a branch is likely to take.
static linking	The linking of procedures at compile time, rather than at link time or at load time.
statically built	Built at compile-time.
stdcx	PPE store double word conditional indexed instruction.
sticky bit	A bit that is set by hardware and remains so until cleared by software.

stmw PPE store multiple word instruction.

storage Any type of data storage, including memory, cache, disk, diskette, and tape. Compare *main storage*.

storage access An access to *main storage* caused by a load, a store, a DMA read, or a DMA write.

storage-access recording The setting of the PTE Reference (R) bit to ‘1’ whenever an attempt is made to read or write the page, and the Change (C) bit to ‘1’ whenever the page is written. The recordings are done by hardware.

store combining The combining of multiple store accesses to storage into a single store access.

STQ Store Queue. The CBE processor has several.

stream A sequence of contiguous memory locations that are accessed by a memory reference throughout the lifetime of a loop (also called a *memory stream*), or a sequence of contiguous register values that are produced by an operation over the lifetime of a loop (also called a *register stream)*.

stream offset The byte-offset of the first desired value in the first register of a stream.

stride The relationship between the layout of an array’s elements in main storage and the order in which those elements are accessed. A stride of length N means that, for each array element accessed, N-1 adjacent memory elements are skipped over before the next-accessed element.

stride-one memory access A memory access pattern in which each element in a list is accessed sequentially.

stswi PPE store string word immediate instruction.

stswx PPE store string word indexed x-form instruction.

stwcx PPE store word conditional indexed instruction.

subword arithmetic operations The conversion of integer arithmetic operations to the equivalent arithmetic operation on narrower-width data types.

subword data type A 2-byte or 1-byte data type, both of which are smaller than a word.

supervisor state The permission level of privileged instructions. The instructions are described in the *PowerPC Operating Environment Architecture, Book III* and are required of software that implements system programs. Compare *problem state*.

superscalar The ability to execute multiple instructions per cycle. It is accomplished with multiple, parallel execution units.

SXU Synergistic Execution Unit. It contains the SPU Odd Fixed-Point Unit (SFS), SPU Even Fixed-Point Unit (SFX), SPU Odd Floating-Point Unit (SFP), and SPU Load and Store Unit (SLS).

symmetric multiprocessing See *SMP*.

sync A PPE and/or SPE synchronize instruction.

synchronous Coordinated in time, with the execution of instructions in an instruction pipeline or among tasks.

synchronization (1) Storage-access ordering for shared-storage environments. (2) A software-enforced alteration of processor state so as to match the sequential ordering of instructions. (3) The use of atomic operations to create semaphores, mutex locks, spin locks, and other synchronization primitives for shared-storage environments. Compare *serialization*. See also *sequential order, program order,* and *context synchronizing*.

system A combination of processors, storage, and associated mechanisms that is capable of executing programs.

system memory See *main storage*.

system storage All program-addressable memory in a system, including main storage (main memory), the PPE's L1 and L2 caches, and the SPE's *local store* (LS). See *main storage*.

table walk Table look-up (table search).

tag group A group of MFC DMA commands. All DMA commands except **getllar**, **putllc**, and **putlluc** are associated with a Tag Group.

taken (1) Said of an interrupt whose service routine is executed. (2) Said of a conditional branch when the condition it is testing is true.

task A *process* (unit of resource ownership) in a multiprogramming (multitasking) environment. A task owns a virtual address space in which it stores processor state, and it may own other resources such as protected access to other processes, I/O devices and files.

TClassID Transfer Class ID.

text segment A segment of programming code.

thread (1) A unit of operating-system scheduling and dispatching that executes sequentially and can be interrupted. Threads are created by processes (tasks), which may own one or more of them, and threads use the resources of the creating process. A thread can be running or waiting to be run.

(2) A sequence of instructions executed within the global context (shared memory space and other global resources) of a process that has created (spawned) the thread. Multiple threads (including multiple instances of the same sequence of instructions) can run simultaneously, if each thread has its own architectural state (registers, program counter, flags, and other program-visible state).

The SPE hardware supports a single thread, per SPE. The PPE hardware supports two threads.

throughput (1) The number of instructions completed per cycle. A high-throughput application design seeks to keep pipelines full. (2) The maximum sustained rate at which a processor can execute an instruction of a particular type, in the absence of any dependencies and assuming infinite caches.

time base The CBE facility that provides the timing functions for the CBE Core-Clock (`NClk`) domain.

TIS Tool Interface Standard.

TKM Token Management Unit. Part of the Element Interconnect Bus (EIB) that software can program to regulate the rate at which particular devices are allowed to make EIB command requests.

TLB Translation Lookaside Buffer. An on-chip cache that translates virtual addresses (VAs) to real addresses (RAs). A TLB caches page-table entries for the most recently accessed pages, thereby eliminating the necessity to access the page table from memory during load/store operations.

TLB hit A TLB access in which the corresponding entry is present and is valid.

tlbie PPE TLB invalidate-entry instruction.

TLBIE request A request for a TLB-entry invalidation that is generated by an SPE's Atomic Unit and broadcast over the EIB.

tlbsync PPE TLB-synchronization instruction.

TMCU Thermal Management Control Unit.

TOC Table Of Contents.

TOE Table of Effective-Address References (EARs). It is used to resolve references from an SPU's address space to the PPU's symbols.

.toe A section in a CESOF object file that contains a TOE. It is generated by the SPU linker and is not seen by application programmers.

TOE shadow An exact copy of a toe segment. In the PPE-ELF object, the TOE shadow is defined in a `.data` section.

token A grant of access to the EIB. When the token manager function is enabled, the SBI will request a token for any memory- or the I/O-bus transaction. After the token is granted, the SBI will send the bus request to the EIB.

touch To cause a cache block to be speculatively loaded. The *PowerPC Architecture* supports instructions that perform this function.

transfer element Same as *list element*.

trap instruction An instruction that tests for a specified set of conditions. If the conditions of a trap instruction are met, the Program Interrupt trap handler is invoked.

trip count The number of iterations in a loop.

TS The transfer-size parameter in an MFC command.

unified cache A cache that stores both instructions and data.

unified register file A register file in which all data types—integer, single- and double-precision floating-point, logicals, bytes, etc.—use the same register file. The SPEs (but not the PPE) have unified register files.

unordered Said of an exception that may generate an interrupt at any time, regardless of the state of the interrupt-processing mechanism.

unroll See *loop unrolling*.

UPC Unified Parallel C. A a parallel-programming extension to the ANSI C language.

update The action, by a load or store instruction, of automatically copying the target address computed by the instruction into the base register used for the address computation. Update instructions are useful for moving repetitively through data structures.

user mode The mode in which problem state software runs. See *problem state*.

V Valid.

VA Virtual Address.

valid A page-table entry or TLB entry that is allocated and correctly associated with the object of its reference.

vector	(1) An instruction operand that consists of a set of data elements packed into a one-dimensional array. The elements can be fixed-point or floating-point values. Most Vector/SIMD Multimedia Extension and SPU SIMD instructions operate on vector operands. Vectors are also called *SIMD* or *packed operands*. (2) See *interrupt vector*.
Vector/SIMD Multimedia Extension	The SIMD instruction set of the *PowerPC Architecture*, supported on the PPE. Also known as AltiVec, which is a Freescale trademark.
VIQ	VSU Issue Queue.
virtual address	An address to the virtual-memory space, which is much larger than the real address space and includes pages stored on disk. It is translated from an effective address (EA) by a segmentation mechanism and used by the paging mechanism to obtain the real address (RA). The maximum size of the virtual-address space is 2^{65} bytes.
virtual memory	The address space created using the memory management facilities of a processor.
virtual mode	The mode in which virtual-address translation is enabled.
virtual vector	A compiler construct that has no alignment constraints and can have any length. Compilers use virtual vectors to carry out preliminary SIMD transformations.
volatile register	A register designated by an ABI as unnecessary to save across procedure calls. Also called a *caller-save register*.
VPN	Virtual Page Number. The number of a page in virtual memory.
VSID	Virtual Segment ID.
VSU	Vector Scalar Unit. In the PPE, the combination of the VXU and FPU.
VXU	Vector/SIMD Multimedia Extension Unit.
W	The write-through bit in a page table entry which controls the processor's accesses to cache and main storage. It is part of the “WIMG” bits.
WAW	A write-after-write dependency.
way	A column of a set-associative, multi-way cache, including a TLB cache. Compare *set*.
weakly consistent storage order	A memory-access model in which the order of processor storage accesses, the order of those accesses with respect to another processor or mechanism, and the order of those accesses in main storage, may be different.
WIMG bits	Four bits in a page table which control the processor's accesses to cache and main storage—“W” is write-through, “I” is caching-inhibited, “M” is memory-coherence, and “G” is guarded.

word Four bytes.

workload A set of code samples that characterizes the performance of the architecture, algorithms, libraries, tools, and compilers.

WPC Write-Port Collision.

wrch SPU write to channel instruction.

writeback A data-cache property that allows modified data to be written only to the cache, rather than also to main storage. The modified data is written to main storage only when the cache line is replaced. Compare *write through*

writeback flag A flag written (for example, by an SPE) to main storage that notifies another processor (for example, the PPE) of an event.

Write With Clean A burst operation caused by a processor executing a **dcbst** instruction or a bus snoop read or clean to a modified block. It is used to tell all lower-level caches that a copy still remains in this level, while updating memory or I/O.

Write With Flush A partial-block write to memory or a sub-block burst operation from the I/O. It is used for caching-inhibited or write-through writes from a processing element.

Write With Kill A burst operation used to tell all snoopers to invalidate any copies of this cache line in their caches, while also storing the line to memory.

write through A data-cache property that requires modified data to be written not only to the cache but also to main storage. Compare *writeback*

XDR Rambus Extreme Data Rate DRAM technology.

XIO Rambus XDR I/O (XIO) cell.

XLAT Translate request queue.

XU Execution Unit. The PPE unit that contains the FXU, LSU, and MMU.

zero-shift policy A reorganization of misaligned data in which each misaligned register stream is shifted to a stream offset of 0 immediately after it is loaded from memory.

Index

Symbols

.bss section, 394
.data section, 394
.spe.elf section, 403, 404
.text section, 394
.toe, 393, 394, 401, 407, 409, 856
.toe section, 393, 394, 401, 403, 409
.toe segment, 403
__builtin_expect, 779
__SPU__, 679
_align_hint, 780
_spe_elf_image, 407

Numerics

16-bit, 57
32-bit, 57
8-bit, 57

A

A, 277, 278, 473
ABI, 402, 823
AC, 88, 823
Address Range, 155
addressing, 79
Addressing Modes, 51, 57
Advanced Encryption Standard, 702
aeneral-purpose registers, 49
AES, 702
aggregation, 646, 647, 655, 824, 838, 849
aligned, 780
aligned reference, 620
alignment, 401, 722
alignment constraints, 620, 644
alignment devirtualization, 648
Alignment interrupt, 251
Allocation Registers, 212
alpha partition, 329
AltiVec, 675, 745, 823
ALU, 724, 823
AOS, 620, 823
API, 391, 392, 599, 823
application, 29, 33, 34, 823
application yielding, 351
argp, 400
arguments, 414
ARPN, 88, 108, 823
array of structures (AOS), 620
arrays, 636
asynchronous, 823
Asynchronous Event Handling, 471
ATO, 107, 823
ATO Flush Collision, 154
atomic access, 117, 166, 282, 823
Atomic cache, 109, 151
atomic commands, 503, 510
atomic operation, 151, 551, 575, 587, 823
atomic synchronization, 575
Atomic Unit, 107, 151, 152
atomic update, 72
atomic-reservation thrashing, 606
attributes, 552
auto-SIMDization, 635
auto-SIMDizing compilers, 637
auto-vectorizing compiler, 637
auto-vectorizing compilers, 637
auxiliary information structures, 397
auxiliary vector area, 398
available to software, 121, 823
AVPN, 823

B

B, 277, 278, 279, 781
bandwidth, 206
Bank Incrementer, 216
banks, 215
barrier, 510, 824
barrier commands, 507, 510, 687
barrier instructions, 554
barrier option, 564
basic block, 624, 639, 644, 646, 666, 689, 824
basic-block aggregation, 646, 655, 824
BClk, 375, 824
beat, 138, 824
BEI, 38, 824
BHT, 135, 691, 824
BIC, 38, 123, 125, 130, 257, 824
BIC Core Clock (BClk), 375
BIF, 38, 161, 162, 266, 824
BIF/IOIF0, 38
big-endian mode, 737
big-endian ordering, 41
bisled, 466
Bit Ranges, 31
BIU, 126, 229, 824
block, 42, 54, 137, 824
blocking, 302, 824
boundedly undefined, 96, 97, 824
branch hint, 665, 688, 689, 691, 825

Branch History Tables, 141
branch-and-link, 488, 824
branch-hint instructions, 691, 692
branch-prediction hints, 760
branch-target buffer (BTB), 692
Broadband Engine Interface (BEI), 38
Broadband Processor Architecture, 826
Broadband Processor Interface, 824
BRU, 298, 711, 752, 825
BTB, 68, 692
BTIC, 691
bubble, 755, 758, 825
Buffers, 682, 686
Built-Ins, 77
Built-ins, 825
bundling, 665
Bus Interface Controller, 257, 824
Bus Interface Unit, 824
bypassing the cache, 138, 825
Byte Ordering, 41
byte substitution, 702
byte-shuffle, 56
ByteSub, 702

C

C, 29, 92, 97, 151, 189, 196, 277, 278, 781, 825
C/C++ Language Extensions (Intrinsics), 744
cache, 45, 48, 182, 184, 825
cache block, 137, 147, 825
cache hint, 760
cache invalidation, 183
cache line, 133, 825
cache touch, 148, 825
cache writebacks, 194
cacheable, 558, 825
cache-line lock, 173
cache-line size, 46, 48
caching inhibited, 552
caching-inhibited, 558, 825
Caching-inhibited (I bit), 91
callee, 398, 825
caller, 398, 489, 825
cast out, 139, 825
castout, 139
CBE Core Clock (NClk), 375, 377, 424, 826, 828, 855
CBE Performance Monitor Interrupt, 273
CBE processor, 33, 826
CBE Slow State, 424
CBE Slow state, 377
CBE task, 42, 826
CBEA, 33, 826
CBEA Embedded SPE Object Format (CESOF), 402
CCF, 375
CCM, 377
Cell Broadband Engine Architecture, 826
Cell Broadband Engine Architecture (CBEA), 33
CESOF, 402, 826
CESOF object layout, 409
CESOF wrapping layer, 406
CFG_TO, 429
Channel Access Facility, 356
channel count, 441, 826
channel instructions, 446
channel interface, 39
channels, 72, 441, 443, 826
Checkstop, 348
CIDR, 156, 826
CIU, 126, 133, 136, 144, 150, 826
CL, 100, 826
Class ID, 154, 155, 156, 158, 444, 455, 513
class ID, 826
classID, 760
ClassID Register (CIDR), 156
clear a reservation, 576
clearing pending events, 467
Clock Domains, 375
code overlays, 681
code partitioning, 616
coherence, 37, 189, 826
coherence domain, 189, 826
coherence granularity, 141, 827
coherency block, 194, 826
columns, 215
combined snoop response, 272, 827
combining, 191
command, 506, 827
Command Issue, 512, 515
committed, 137, 827
common subexpression elimination, 666, 827, 845
Compatibility with PowerPC Code, 729
compiler, 637
Compiler Directives, 779
compiler optimizations, 757
complete, 52, 75, 827
completion, 514
Composite Intrinsics, 77
Configuration Ring, 121, 123, 207, 228, 230
Configuration-Ring Settings, 429
congruence class, 93, 108, 168, 184, 827
constant folding, 665, 827
contex switch, 351
context, 351, 827
Context Save Area (CSA), 352
context switch, 827
context synchronizing, 238, 247, 274, 827
control plane, 36, 827
conversion of data length, 659
converting scalar data, 620
core clock, 375, 827
Core Clock Frequency (CCF), 375

Core Clock Multiplier, 377
Core Clock Multiplier (CCM), 377
Core Stop Safety, 428
CORE_CLK, 375
count register, 50, 828
count_cycles field, 781
CPI, 699, 700, 704, 706, 828
CPL, 265, 828
CR, 50, 670, 828
CR6, 670
critical quadword, 145
critical section, 600
critical-sector first, 138
CSA, 352, 828
CSI, 711, 758, 828
CTR, 50, 828
cumulative ordering, 567, 828
curly braces style, 675
Current Priority Level, 265
cycle, 35, 38, 828

D

DABR, 332, 340
DABRX, 332, 340
DAR, 241, 249, 250, 252, 828
data, 34, 35, 828
data beat, 138, 145, 828
data effective-to-real address translation (D-ERAT), 724
Data ERAT, 83, 829
data ERAT, 150
data hazard, 50, 753, 828
data length, 658
data plane, 35, 828
Data Segment interrupt, 249
Data Storage interrupt, 248
data stream, 147, 148, 828
Data Types, 57, 59, 744
data types, 51, 59, 639, 744, 774
Data-level parallelism, 55
data-plane, 36
DCache, 135, 136, 828
dcbf, 144, 146, 149, 197, 209, 211, 248, 577, 828
dcbst, 146, 148, 149, 211, 248, 577, 828
dcbt, 139, 146, 147, 148, 149, 209, 828
dcbtst, 146, 147, 148, 149, 209, 577, 828
dcbz, 144, 145, 146, 248, 576, 828
deadlock, 589, 828
DEC, 383, 828
Decrementer, 384, 447
decrementer, 480, 497, 829
Decrementer interrupt, 254
Decrementers, 383
demand fetch, 138, 829
demand-paging, 79
denormalized number, 829
denormals, 59, 68
dependence, 636, 638, 640, 666, 829
dependences, 636, 666
dependencies, 758, 759
D-ERAT, 724, 829
DERR, 112, 117, 119, 247, 282, 829
development tools, 29, 33
Device Memory, 829
device memory, 557
devirtualization, 648, 653, 656, 661, 663
DFQ, 135, 150, 829
DigFiltDly, 429
Digital Thermal Sensors (DTSs), 426
DIQ, 757, 829
Direct Memory Access, 34, 829
Direct Memory Access Controller, 73, 829
Directives, 779
directives, 611, 664
Disableable, 238
dispatch block, 755
displacement, 51, 829
Distributed-Memory Processing, 609
divide and conquer, 603
DLQ, 135, 150, 829
DMA, 34, 505, 829
DMA buffers, 687
DMA command, 507, 508, 517, 829
DMA Command Parameter Registers, 514
DMA commands, 72, 507
DMA Data Transfer Commands, 72
DMA list, 509, 829
DMA queue, 517, 829
DMA Transfer, 516
DMA transfer, 34, 35, 505, 516, 521, 830
DMAC, 73, 829
DMA-List Command, 527, 829
DMA-List Command Stall and Notify Event, 484
DMA-List Command Stall-and-Notify Event, 499
DMA-List Commands, 509
documentation conventions, 30
dominant-shift policy, 651
dominant-shift reorganization, 830
Double Buffering, 682, 683
doubleword, 830
DP, 70, 76, 770, 830
DPFE, 135, 136, 148, 150, 830
DR, 83, 248, 303, 830
DRAM memory, 38
DSI, 95, 111, 115, 117, 119, 241, 280, 282, 284, 288, 289, 830
DSISR, 241, 249, 250, 251, 252, 303, 830
DTS, 426, 830
DTSs, 426
dual-issue, 687, 688, 768, 830
dynamic branch prediction, 694, 696, 830

dynamic linking, 412, 830
Dynamic Thermal-Management registers, 432

E

E, 166, 167, 781
EA, 47, 830
ea_value, 408
eager shift, 650, 830
eager-shift policy, 650
EAH, 528, 830
EAR, 394, 407, 830
ECC, 38, 48, 64, 142, 830
Effective Address, 40, 47, 79, 830
Effective Address Entry (EAR) structure, 394
Effective Address Reference (EAR) structure, 407, 830, 855
effective-address space, 39
Effective-to-Real Address Translation, 135, 831
EIB, 37, 135, 831
EIB Possible Livelock Detection Interrupt, 272
eieio, 555, 565, 566, 831
elem_slide, 649
elem_splat, 649
Element Interconnect Bus, 257, 831
Element Interconnect Bus (EIB), 37
ELF, 402, 831
ELF header, 392, 831
ELF object files, 392
Enableable, 238
Endian Order, 32
endian order, 41
Endian Support, 737
entry point, 395, 399
environment note, 394
envp, 400
ERAT, 80, 81, 83, 135, 831
error handler, 237
Error-Correcting Code, 830
Errors, 517
ESID, 85, 831
even pipeline, 687, 831
Event Handling Protocols, 473
Event-Control Channels, 464
Event-Handling Protocols, 470
Events, 463
events, 438, 464, 781
exception, 237, 831
exception handler, 237
Exceptions, 237
executables, 391
execution order, 551, 558
execution synchronizing, 556, 831
external access, 558, 559, 831
external events, 463
External Interrupt, 251
External Interrupt Controller, 291
External Interrupts, 262
External Time-Base Sync Mode, 381

F

MSR, 757
fence, 507, 831
fence option, 564
fenced commands, 687
fetch, 768, 831
fetch group, 688, 768, 769, 832
Fields, 31
FIFO, 146, 532, 752, 832
FIR, 265, 277, 832
fixed-point, 47, 50, 51, 832
FlexIO, 38, 161, 832
FlexIO_0, 161
FlexIO_1, 161
FLIH, 487, 832
Floating-Point Unavailable interrupt, 254
Floating-point unit, 299, 832
flush, 832
flush condition, 755
flush point, 137
Flush transaction, 832
flushes, 758
Fmax, 377
for loop construct, 635
fork and join, 603
fork and join model, 612
FP, 303, 832
FPR, 49, 832
FPSCR, 50, 67, 373, 670, 832
FPU, 46, 832
frequency, 377
fres, 736, 832
fscrrd instruction, 69
fscrwr instruction, 69
function-inlining, 689
FXU, 47, 832

G

G, 88, 91, 832
garbage collector, 498
gather, 638, 848
gather and scatter, 510
GCD, 647, 832
general-purpose registers, 832
Generic Intrinsics, 60, 77, 832
get, 508, 509
get_ps, 833

getb, 509
getbs, 509
getf, 509
getfs, 509
getl, 509
getlb, 509
getlf, 509
getllar, 510, 588, 833
gets, 509
GPR, 49
GPRs, 67, 833
granularity, 141, 577
Graphics Rounding Mode, 59, 742
graphics rounding mode, 675
guaranteed latency, 296
guarded, 552, 833
Guarded (G bit), 91
H, 88, 168, 169, 170, 171, 172, 277, 278, 279, 833

H, I, J, K

Hardware Environment, 37
harvest, 358
Harvesting an SPE, 358
hashed page table, 81, 87, 833
hazard, 138, 139, 688, 753, 833
HBR, 693
hbrp, 772
hcall, 329, 833
HDEC, 328, 332, 383, 833
header, 394
heavyweight sync, 554
HID, 112, 253, 332, 833
high, 30, 833
hint (H) bit, 170, 173
hint for branch, 691
hint for branch (HBR) instructions, 693
hint stall, 692, 833
hint-for branch instructions, 691
hint-trigger address, 691, 692
HL, 167
hoist, 692, 833
hrfid, 331
HRMOR, 332, 335, 833
HSPRG0, 332
HSPRG1, 332
HSRR0, 331
HSRR1, 331
HTAB, 87, 833
HTABSIZE, 103
HV, 290
HW, 167
hypervisor, 245, 290, 328, 733, 834
hypervisor call, 329, 834
Hypervisor Decrementer interrupt, 254
Hypervisor Interrupts, 291
Hypervisor State, 301, 331
I, 91, 196, 834
I/O access, 189, 835
I/O address, 164, 835
I/O address translation, 176
I/O Address Translation Interrupt, 271
I/O architecture, 161
I/O device, 165, 182
I/O Device Interrupt, 270
I/O devices, 38, 65, 91, 121, 126, 161, 836
I/O Exception Status Register (IOC_IO_ExcpStat), 176, 182
I/O exceptions, 181
I/O Identifier, 189
I/O interface considerations
memory-mapped I/O interface operations, 92
I/O Interface Controller (IOC), 162, 203, 836
I/O interfaces, 38, 161
I/O operation, 189, 836
I/O Page Table (IOPT), 171
I/O Segment Table (IOST), 169
I/O Segment Table Origin (IOC_IOST_Origin) register, 166, 345
I/O storage model, 189
I/O unit, 189, 836
I/O-address space, 345
I/O-hosting partition, 329
I0, 757
I1, 757
IBuf, 133, 298, 752, 834
ICache, 135, 136, 834
icbi, 138, 146, 147, 248, 834
ID, 41, 83, 85, 96, 98, 104, 118, 182, 205, 263, 266, 834
IE, 449
IEEE 754, 68, 675, 834
I-ERAT, 83, 834
IFAR, 299, 834
if-then-else statements, 636
IGP, 268, 834
IIC, 237, 240, 834
ILP, 640, 834
ILP32, 606, 834
immediate operand, 691, 834
imprecise interrupt, 238, 835
inbound, 191
inbound access, 189, 835
incoming IOIF, 195
index, 52, 83, 87, 93, 835
index vector, 638
indirect, 449, 463, 468, 835
inexact result, 670
infinities, 59
Initial Machine State, 395, 399
initialization, 395, 414

inline assembly, 779
inline expansion, 835
inlining, 689
inner loops, 646
in-order, 48, 53, 551, 558, 688, 768, 835
Instruction and Data Relocate mode, 332
instruction buffer, 136
Instruction ERAT, 83, 834
instruction fetch, 768
instruction runout, 771, 835
instruction scheduling, 665
Instruction Segment interrupt, 251
Instruction Storage interrupt, 250
Instruction Types, 52, 57, 58
instruction-fetch buffer, 665
instruction-fetch starvation, 665
Instruction-level parallelism, 834
instruction-sequencing error, 835
Internal Interrupt Controller (IIC), 237, 834
Internal Time-Base Sync Mode, 378
Interprocessor Interrupt, 268
InterProcessor Interrupt (IPI), 240, 262
interrupt, 237, 245, 835
Interrupt Address Save/Restore Channels, 469
Interrupt Controller (IIC), 240
Interrupt Generation Port (IGP), 268
interrupt handler, 237, 487
Interrupt Handling, 242
interrupt handling, 493
Interrupt Latencies, 289
interrupt packet, 196, 262, 269, 285, 835
Interrupt Pending Port (IPP), 264
Interrupt Priorities, 287
interrupt protocol, 487
Interrupt Registers, 241, 263, 278
interrupt service routine, 237
Interrupt Stack, 495
interrupt vector, 242, 243, 835
Interrupts, 195, 237, 287, 471
intervention, 142, 166, 210, 835
intrinsics, 59, 60, 744, 835
invalid SIMDization, 640
invalidation, 183, 187
inverted mapping, 79
inverted page table, 79, 835
IOC, 162, 203, 836
IOC Base Address Registers, 175
IOC_BaseAddr0, 175
IOC_BaseAddr1, 175
IOC_BaseAddrMask0, 175
IOC_BaseAddrMask1, 175
IOC_IO_ExcpStat, 176, 182
IOC_IOST_Origin, 166, 345
IOID, 165, 166, 174, 189, 836
IOID field, 172
IOIF, 161, 836
IOIF device, 182, 836
IOIF in bus, 195
IOIF out bus, 195
IOIF protocol, 161, 836
IOIF0, 161
IOIF1, 161, 836
IOIFs, 161
IOPT, 171, 836
IOPT Base RPN field, 170
IOPT cache, 184
IOST, 169, 836
IOST cache, 182
IOST Origin, 167
IOST Size, 167
IPC, 53, 58, 711, 712, 738, 836
IPFQ, 133, 135, 150, 836
IPI, 240, 262, 268, 836
IPP, 264, 836
IQ0, 514
IQ1, 514
IQ2, 514
IR, 83, 836
IS, 449, 836
IS field, 734
ISA, 74, 761
ISEG, 251, 836
ISI, 250, 836
ISRC, 266, 267, 836
Issue, 515, 521, 688
Issue Rules, 750
isync, 556
IU, 46, 135, 836
Java-mode, 670
K, 837
key, 342, 500, 837
Kp, 86, 100, 248, 837
Ks, 86, 100, 248, 837

L

L, 86, 87, 88, 90, 837
L1, 135, 136, 837
L2, 135, 141, 837
Large Pages, 88, 107
LARX-reserve transaction, 837
Latency, 53, 58, 235, 711, 712, 738
latency, 75, 296, 837
lazy-shift policy, 650, 837
ld, 712, 728, 837
ldarx, 575, 837
ldbrx, 729, 731
Le, 474
LEAL, 528, 837
least-recently used, 94, 109, 138, 837
least-significant, 30, 837

length conversion, 652, 659, 661, 837
length conversions, 648
length devirtualization, 653, 656
Level-1 cache memory, 136, 837
Level-2 cache memory, 141, 837
LG, 277, 278, 279
lightweight sync, 554, 555
linear thermal diode, 426
linkable objects, 404
Linked Data Structure Traversal, 500
linked list, 296
linker, 392, 394, 401, 403, 407, 413, 838
list element, 73, 527, 838
List Size, 454
List Stall-and-Notify, 459
List Transfer Size, 528
List-Element Effective Address Low, 528
little-endian order, 52
livelock, 570, 590, 838
LMQ, 135
lmw, 252, 712, 838
lnop, 693, 765, 838
load combining, 191, 838
load-combining, 555
loader, 392, 403, 413, 414, 838
loader parameters, 415
loading, 391
local CBE processor, 211
local storage, 34, 838
local storage domains, 39
local store, 34, 838
Local Store Address, 507, 839
Local Store Context Save Area (LSCSA), 352
lock, 143, 152, 154, 158, 166, 167, 168, 173, 461, 465, 476, 503, 510, 838
locked, 158
Lock-Line Reservation Lost Event, 476, 503
logical partition, 291, 327, 733, 838
Logical Partition Control Register, 839
Logical Partition ID, 839
logical partitioning, 327
Logical Partitioning Control Register (LPCR), 733
logical PPE, 838
loop aggregation, 646, 655
loop bounds, 635, 654
loop collapsing, 666
loop distribution, 666, 838
loop fusion, 666
loop interchange, 666, 838
loop rerolling, 666
loop unroll-and-pack, 666
loop unrolling, 666, 689, 838
loop versioning, 666, 839
loopback, 272
loop-carried dependences, 666
looped-back operations, 174, 839
loop-level aggregation, 647, 838
loops, 646, 666
low, 30, 839
LP, 86, 88, 277, 278, 279, 839
LP64, 606, 839
LPAR, 254, 291, 733, 839
LPCR, 332, 733, 839
LPES, 335, 839
LPID, 336, 839
LPIDR, 331
LR, 49, 839
Lr, 473
LRU, 94, 109, 138, 839
LRU algorithm, 184
LS, 39, 64, 839
LS addresses, 687
LS mappings to main storage, 557
LS memory attributes, 557
LSA, 127, 507, 839
LSCSA, 352, 839
LSU, 46, 135, 738, 839
LSZ, 839
LTS, 528, 839
lvlx, 707
lvrx, 707
lwarx, 575, 839
lwsync, 555, 565, 566

M

M, 88, 91, 114, 172, 189, 196, 277, 278, 279, 839
m, 81, 839
M:N thread model, 841
Machine Check, 290, 348
Machine Check interrupt, 246
Machine State Register, 841
mailbox, 72, 505, 840
Mailboxes, 531
mailboxes, 461, 531
main memory, 566, 840
main storage, 34, 39, 40, 840
Main Thread, 42
mangle, 393, 840
mangled, 393
many-to-one signalling, 544
Mapping PPE to SPEs, 674
Maskable, 238
master and slave, 603
master thread, 612
MaxSlowModeNclkDivider, 377
MB, 840
Mb, 474
MC, 324, 711, 840
ME, 290, 332
Me, 474

Mediated External exception, 273
Mediated External Exception mode, 338
Mediated External Interrupt, 273, 275
Mediated External interrupt, 273
Mediated Interrupt (MER), 733
Memory, 34, 840
memory- and cache-coherence, 37
memory banks, 215
memory coherence, 173, 189
Memory coherency (M bit), 91
Memory Flow Controller, 70, 840
Memory Interface Controller, 38, 841
Memory Management Unit, 47, 841
memory stream, 627, 648, 840
memory-channel 0, 216
memory-channel 1 tokens, 216
memory-coherent, 552
memory-mapped, 32, 840
memory-mapped I/O (MMIO), 32, 121
MER, 339, 733
MERSI, 142, 840
MF, 277, 278, 279
MFC, 840
MFC Class ID channel, 455
MFC Command Opcode channel, 455
MFC Command Queue, 507
MFC Command Tag Identification channel, 455
MFC Commands, 506, 840
MFC Effective Address High channel, 453
MFC Effective Address Low or List Address channel, 453
MFC Interrupts, 268, 276
MFC Local Storage Address channel, 452
MFC Multisource Synchronization Facility, 448, 553, 567, 568
MFC Multisource Synchronization Register, 568
MFC Pause State, 426, 840
MFC Proxy Command Queue, 520
MFC proxy commands, 840
MFC Read Atomic Command Status channel, 461, 589
MFC Read List Stall-and-Notify Tag Status channel, 459
MFC Read Tag-Group Query Mask channel, 457
MFC Read Tag-Group Status channel, 458
MFC SPU Command Queue Available Event, 483, 502
MFC synchronization commands, 562
MFC Tag-Group Management Channels, 456
MFC Transfer Size or List Size channel, 454
MFC Write List Stall-and-Notify Tag Acknowledgment channel, 460
MFC Write Multisource Synchronization Request Channel, 570
MFC Write Multisource Synchronization Request channel, 448
MFC Write Tag Status Update Request channel, 457, 458
MFC Write Tag-Group Query Mask channel, 457
MFC_Cmd, 483
MFC_EAH, 453
MFC_EAL, 453
MFC_LSA, 452
MFC_MSSync, 568
MFC_RdAtomicStat, 461, 588, 589
MFC_RdListStallStat, 459
MFC_RdTagMask, 457
MFC_RdTagStat, 458, 523
MFC_SDR, 103
MFC_Size, 454
MFC_TagID, 455
MFC_WrListStallAck, 460
MFC_WrMSSyncReq, 448, 570
MFC_WrTagMask, 457
MFC_WrTagUpdate, 457
mfceieio, 510, 841
mfcsync, 510, 562, 841
mfspr, 841
MIC, 38, 841
MIC Auxiliary Trace Buffer Full Interrupt, 270
MIC Clock (MiClk), 375
MiClk, 375, 841
microarchitecture, 607, 841
microcoded instructions, 723
microthreads, 498
MIMD, 609
MinStopPPE, 429
MinStopSPE, 429
misaligned data, 638
misalignment, 648
misalignments, 654
mispredicted branches, 689
miss handler, 614
miss penalty, 841
mixed-mode SIMDization, 663, 841
MMIO, 32, 121
MMU, 47, 135, 841
MOD, 841
MODINTV, 841
most-significant, 30, 841
MPI, 609, 841
Ms, 473
MSb, 30, 841
MSR, 290, 331, 841
MT, 216, 217, 841
mtmsr, 331, 719, 841
mtspr, 96, 97, 101, 156, 253, 315, 382, 384, 712, 841
multibuffering, 682, 685
Multiple Events, 473
multiplies, 68, 761
multiprocessing, 609
Multisource Synchronization, 448
Multisource Synchronization Event, 474, 570, 572, 574
multithreading, 608, 842
mutex lock, 503, 600, 823, 842

mutual exclusion, 600
mutual-exclusion (mutex) lock, 823
mutual-exclusion lock, 503

N

N, 86, 88, 100, 103, 104, 434, 842
n, 81, 842
NaNs, 59, 69, 842
NClk, 375, 377, 424, 828, 842
NCU, 126, 133, 135, 136, 842
Non-Cacheable Unit, 206, 229, 842
non-volatile variables, 687
no-op, 385, 733, 842
No-Op forms, 733
nop, 302, 316, 842
Not-a-Number, 69, 842
Notation, 30
NPC, 414, 842
NPPT field, 170
NUMA, 335, 842

O

object files, 392
object module, 391, 842
object-file formats, 391
odd pipeline, 687, 842
offset, 99, 124, 842
OGSA, 610, 842
one-to-one signaling, 544
OpenMP, 611, 664, 842
operating system, 29, 33, 42
optional PowserPC instructions implemented, 736
OR mode, 544
order, 551, 558, 687
order of storage, 190
ordered, 551, 842
OS, 329, 842
out of order, 551, 842
outbound access, 189, 843
outgoing IOIF, 195
out-of-order execution, 512
overlay, 403, 406, 843
overwrite mode, 544

P

p, 81, 97, 843
pack, 661
packed data, 639
packed operands, 55, 619, 857
page, 79, 843
page fault, 87, 843
page protection (PP) field, 172
page sizes, 74
page table, 79, 87, 107, 843
Page Table Entry, 846
Page Table Entry (PTE), 87
page tables, 79
page-history recording, 843
page-size (PS) field, 171
page-table entry group, 843
paging, 79
parallel for construct, 664
parallel memory operations, 216
Parallel Programming, 599
parallel_reduct, 649
parallel-array form, 621
parallelism, 55, 601, 602, 603, 639
parallelization, 551
parenthesis style, 675
parity, 46, 47, 104, 107, 118, 142, 843
partial stores, 653
partial-copy operations, 665
partition manager, 616
partitions, 327
path length, 695, 843
PC, 34, 498, 843
Performance, 385, 530
performance, 35
Performance Counters, 438
Performance Monitor Interrupt, 273
performance monitor signals, 781
Performance Monitoring, 437
Performance Monitoring Events, 438
performance simulation, 437, 843
permute, 56
Pervasive Logic, 843
pin TLB entries, 207
pinned, 356, 843
pipeline, 75, 687, 752, 761, 768, 844
pipeline 0, 687
pipeline 1, 687
pipeline stages, 53, 752
pipelined hint mode, 693, 844
PIR, 331, 844
PLL, 377, 844
PLL Multiplier, 377
PLL Multiplier Setting, 377
PLL Reference Clock (PLL_REFCLK), 377
PLL_REFCLK, 377
PLLmultiplier, 377
plugins, 393, 395, 844
pmN_control registers, 781
pointer arithmetic, 636
pointer-chasing, 296
Polling, 470
polling, 237

POR, 121, 123, 207, 228, 229, 245, 329, 844
POSIX, 503, 579, 617, 844
POSIX threads library, 617, 844
Power and Thermal Management, 423
Power Processing Storage Subsystem, 438
Power Processor Unit, 438, 844
Power-Management States, 423
Power-On Reset (POR), 121, 123, 228, 229, 245, 329
PowerPC, 33, 844
PowerPC 970, 45, 844
PowerPC Architecture, 33, 45, 844
PowerPC Compatibility, 729
PowerPC Extensions, 729
PowerPC instructions not implemented, 736
PowerPC Processor Element (PPE), 45
PowerPC Processor Storage Subsystem (PPSS), 48
PowerPC Processor Unit (PPU), 46
PowserPC instructions, 736
PP, 81, 88, 100, 111, 118, 172, 844
PPE, 45, 844
PPE Pause (0) State, 425, 844
PPE registers, 50
PPE's SPU Channel Access Facility, 356
PPSS, 48, 135, 844
PPU, 46, 135, 844
pragma, 599, 844
precise interrupt, 238, 845
precise trap, 69
precision, 675
predicate, 60, 503, 671, 696, 749, 845
Predicate Intrinsics, 60
Predication, 689
predictive commoning, 666, 845
preemptive context switch, 351
preferred scalar slot, 74, 706, 845
preferred slot, 74, 845
prefetch, 48, 64, 66, 91, 126, 133, 134, 135, 136, 137, 142, 144, 145, 148, 150, 845
prefix, 60
prependW, 649
primary partition, 329
primitive, 551, 576, 577, 580, 845
Priorities, 287
Privilege 1, 127, 340, 341, 845
Privilege 2, 127, 340, 845
Privilege 3, 127, 340, 845
Privilege States, 301
Privileged Attention Event, 476, 503
privileged attention event, 358
privileged software, 845
Privileged State, 301, 331, 845
privileged state, 331
privilege-state programming, 29
Problem State, 301, 331, 845
problem-state programming, 29
problem-state registers, 50, 67
process-management primitives, 391
profiling, 498
program header, 392, 846
Program interrupt, 252
program order, 48, 552, 846
Programming, 39
programming-environment overview, 29
PRV, 121, 846
PS, 170
pseudo-LRU, 94, 109, 138, 846
PT_NOTE, 394
PTE, 87, 107, 846
PTE Groups (PTEGs), 89
PTEG, 89, 846
push, 142, 232, 846
put, 508
putb, 508
putbs, 508
putf, 508
putfs, 508
putl, 509
putlb, 509
putlf, 509
putllc, 510, 588, 846
putlluc, 510, 589, 846
putqlluc, 510, 589
puts, 508

Q

QoS, 38, 235, 846
quadword, 57, 846
Quality of Service, 235, 846
Qv, 474

R

R, 81, 88, 92, 97, 103, 114, 116, 151, 846
RA, 47, 846
RAG U, 214, 846
RAGs, 203, 205, 846
RAID, 205, 846
RAM, 203, 846
range hit, 156, 846
Range Mask Register (RMR), 155
Range Start Register (RSR), 155, 156
RAS, 125, 846
Rate Decrementer, 218
RAW, 753, 756, 759, 846
RC, 113, 134, 145, 150, 846
Rc, 363, 723, 846
RC machine, 152, 154, 847
RC update, 151, 847
RC4, 706

rchcnt, 442, 446, 847
RClassID, 513, 847
RclassID, 98, 154, 155, 156, 157, 455
rdch, 446, 847
real address, 99, 847
real address space, 121
real addressing mode, 80, 99, 333, 847
Real Addressing Mode facility, 334
Real Mode, 334
real mode, 80, 83, 99, 847
Real Mode Limit Select (RMLS) field, 733
Real Mode Offset, 334
Real-Mode Address Boundary Facility, 116
record bit, 723, 847
recording, 723, 847
recording form, 670
recording forms, 672
recording instructions, 53
Recoverable Interrupt, 290
reduct, 649
RefDiv, 377
Referencing Registers, 31
refill window, 665
Register Initialization, 396, 399
register spill, 847
register stream, 627, 648, 847
registers, 50, 67
Related Publications, 29
relocation, 79, 847
Replacement Class ID, 154, 155, 455, 513
Replacement Management Table, 847, 848
Replacement Management Table (RMT), 98, 110, 154, 158
replacement-management tables, 48
requesters, 203, 204, 205
requestors, 38
reservation, 575, 847
reservation clearing, 576
reservation granule, 575, 847
reserved, 121, 125, 848
Reserved Regions, 32
resolve, 319, 342
Resource Allocation Group, 846
Resource Allocation Groups (RAGs), 203, 205, 846
Resource Allocation ID, 205, 846
Resource Allocation Management (RAM), 203, 846
resources, 203
restrict, 779
retire, 75, 827, 848
Revision Log, 875
RI, 290, 848
RLD, 135, 136
RMLR, 99, 116, 848
RMLS, 335, 733
RMO, 338, 848
RMOR, 332, 334, 848
RMQ, 135
RMR, 155, 848
RMSC, 99, 100, 336, 848
RMT, 98, 110, 154, 158, 848
round towards zero, 68
rounding, 68, 742
rounding mode, 59
rows, 215
RPN, 170, 172, 173, 848
RPN field, 172
RSR, 155, 156
runout, 771, 848
runtime alignment, 654, 848
Runtime Environment, 42
runtime loader, 413, 414
runtime partition manager, 616

S

S, 119, 173, 190, 192, 196, 277, 278, 279, 461, 781
S1, 473
SA0, 514
SA1, 514
SA2, 514
saturation, 670
SBI, 112, 279, 514, 848
scalar, 51, 63, 74, 848
scalar loads and stores, 705
scatter-gather, 73, 296, 510, 527, 638, 829, 848
scheduling, 665
SCN, 68, 848
sdbrx, 730, 732
sdcrf, 153, 513, 848
sdcrt, 153, 513, 849
sdcrtst, 153, 513, 849
sdcrz, 119, 153, 284, 513, 849
SDR, 81, 89, 103, 849
SDR1, 127, 332
SE, 277, 278
section header, 392, 849
sections, 392, 393
segment, 79, 849
segment buffer, 79
Segment Lookaside Buffer, 283, 850
segmentation, 79
segments, 392, 404
select-bits (selb) instruction, 689
select-bits intrinsic, 689
semaphore, 503, 823, 849
send-signal commands, 510
SenSampTime, 429
sequential order, 849
serialization, 590, 849
set, 92, 107, 849

SF, 277, 278, 279
SFP, 68, 849, 853
SFS, 68, 76, 849, 853
SFX, 68, 849, 853
shared data structures, 606
Shared-Memory Processing, 609
shift, 649, 650, 651
short-loop aggregation, 849
SHT_NOTE, 394
signal, 414, 542, 850
signal notification, 72, 542, 850
signal type, 421
Signal-Notification 1 Available Event, 477
Signal-Notification 2 Available Event, 478
signal-notification channels, 447, 543
signals, 72, 438, 781
SIMD, 55, 850
SIMD operands, 55, 619
SIMD operations, 56
SIMD programming, 619
SIMDization, 57, 619, 637
SIMDization epilog, 630
SIMDization phases, 644
SIMDization prolog, 630
SIMDize, 620, 850
Single Instruction Multiple Data, 619, 850
Single-Instruction, Multiple-Data (SIMD) Vectorization, 55
single-ported, 64, 67, 850
single-step operation, 414
skipW, 649
SLB, 47, 74, 135, 850
SLB entry (SLBE), 85
SLB Mapping, 106
SLB_ESID, 105
SLB_Index, 105
SLB_Invalidate_All, 106
SLB_Invalidate_Entry, 105
SLB_VSID, 105
SLBE, 85, 850
slbia, 84, 86, 105, 714, 736, 850
slbie, 84, 105, 714, 736, 850
slbmfee, 86, 105, 714, 726, 850
slbmfev, 86, 105, 714, 726, 850
slbmte, 86, 105, 714, 726, 850
SLIH, 487, 850
Slot 0, 512
Slot 1, 512
Slot alternation, 514
slow mode, 38, 377, 850
Slow state, 377
Slow-mode dividers (SMDs), 377, 850
SLS, 68, 850, 853
SMD, 377, 850
SMM, 41, 74, 850
SMP, 37, 48, 162, 850
Sn, 474
sndsig, 510, 850
sndsigb, 510, 851
sndsigf, 510, 851
snoop, 48, 851
snoop push, 851
snoop-write queue, 568
SNP, 135, 136, 851
SO, 192, 196
SO field, 172
SOA, 620, 851
software cache, 611, 613, 851
software monitor, 600
software pipelining, 851
software-managed cache, 851
software-managed memory, 851
southbridge, 240, 851
SP, 70, 75, 400, 414, 489, 770, 771, 851
SPE, 63, 851, 852
SPE Interrupts, 276
SPE loading, 391
SPE LS memory attributes, 557
SPE Pause, 851, 852
SPE registers, 67
SPE thread, 42, 851
spe_program_handle data structure, 405
Special Purpose Register, 852
Specific Intrinsics, 60, 77, 851
SPE-ELF Environment Note, 394
SPE-ELF Name Note, 395
SPE-ELF objects, 393
spin lock, 356, 578, 581, 590, 593, 600, 852
splat, 58, 738, 852
SPR, 86, 253, 263, 376, 719, 852
SPU, 63, 851, 852
SPU Channel Access Facility, 356
SPU Channel and DMA Unit (SSC), 76
SPU Code Performance, 385
SPU Decrementer Event, 480
SPU Decrementer event, 497
SPU Event-Management Channels, 447
SPU Floating-Point Unit (SFP), 68
SPU Instruction Set Architecture (ISA), 74
SPU Interrupts, 268
SPU ISA, 74, 761, 852
SPU Mailbox Status register, 539
SPU Odd Fixed-Point Unit (SFS), 76
SPU Pause State, 426
SPU Read Decrementer channel, 448
SPU Read Event Mask, 468
SPU Read Event Mask channel, 468
SPU Read Event Status, 466
SPU Read Inbound Mailbox, 538
SPU Read Inbound Mailbox Available Event, 482, 502
SPU Read Inbound Mailbox channel, 538
SPU Read Machine Status channel, 449
SPU Read State Save-and-Restore channel, 450

SPU Signal Notification 1 channel, 544
SPU Signal Notification 2 Channel, 544
SPU Signal Notification 2 channel, 544
SPU Signalling Channels, 447
SPU Signal-Notification Available Event, 502
SPU target definition, 679
SPU Write Decrementer channel, 384, 447, 448
SPU Write Event Acknowledgement, 467
SPU Write Event Mask, 467
SPU Write Event Mask channel, 467
SPU Write Outbound Interrupt Mailbox Available, 480
SPU Write Outbound Interrupt Mailbox channel, 534
SPU Write Outbound Mailbox Available Event, 479
SPU Write Outbound Mailbox channel, 533
SPU Write State Save-and-Restore channel, 450
spu_env structure, 394
SPU_Mbox_Stat, 539
spu_mffpscr intrinsic, 69
spu_mtfpscr intrinsic, 69
spu_program symbol, 405
spu_program_handle data structure, 409
SPU_RdDec, 384, 447, 448
SPU_RdEventMask, 468
SPU_RdEventStat, 466
SPU_RdInMBox, 482
SPU_RdInMbox, 538
SPU_RdMachStat, 449
SPU_RdSigNotify1, 544
SPU_RdSigNotify2, 544
SPU_RdSSR0, 450
SPU_WrDec, 384, 447, 448
SPU_WrEventAck, 467
SPU_WrEventMask, 467
SPU_WrOutIntrMbox, 480, 534
SPU_WrOutMbox, 479, 533
SPU_WrSSR0, 450
SPU-ELF executable object, 409
spuid, 400
SRR0, 852
SRR1, 852
SSC, 68, 76, 852
SSE, 29, 852
stack frame, 396, 400
Stack Initialization, 396, 400
stale, 139, 344, 466, 475, 537, 852
stall, 761, 852
stall points, 755
stallable, 693
stall-and-notify flag, 459, 484
Stalling, 470
stalling, 302
stalls, 758
starvation, 665, 852
static branch prediction, 141, 694, 852
static linking, 413, 852
statically built, 141, 328, 402, 407, 852
Status, 517
stdcx, 575, 852
STI, 33
sticky bit, 69, 852
stmw, 252, 715, 722, 727, 853
stop-and-signal, 414
storage, 32, 34, 39, 853
storage access, 551, 853
storage alignment, 722
storage barriers, 505
Storage Control Attributes, 91
storage domains, 39
storage model, 91, 189, 551
storage order, 173, 190
storage-access recording, 853
storage-control attributes, 552
store combining, 191, 555, 853
STQ, 135, 136, 853
stream, 605, 627, 853
stream offset, 648, 853
streams, 648
stream-shift operations, 627
stride, 636, 644, 853
stride-one accesses, 636
stride-one memory access, 853
stride-one memory accesses, 644
strongly ordered transfer model, 685
structure of arrays (SOA), 620
stswi, 252, 713, 715, 727, 853
stswx, 252, 713, 715, 727, 853
stvlx, 653, 707, 738, 742
stvrx, 653, 707, 738, 742
stwcx, 142, 152, 248, 249, 252, 319, 575, 577, 587, 588, 715, 756, 758, 853
subword arithmetic, 665
subword arithmetic operations, 664, 853
subword arithmetic optimizations, 664
subword data type, 664, 853
superscalar, 607, 608, 853
supervisor mode, 29
supervisor state, 36, 81, 331, 853
SXU, 63, 853
symbol tables, 392
symbols, 393, 840
symbols used, 31
symmetric multiprocessing, 35, 37, 162, 854
sync, 554, 565, 566, 854
Sync Modes, 378, 381
synchronization, 448, 551, 854
Synchronization Commands, 510
synchronization commands, 507, 562
synchronization instructions, 560
synchronization primitives, 577
synchronization variables, 606
synchronizing, 238
synchronizing events, 605

synchronous, 854
Synchronous Event Handling, 470
Synergistic Memory Management Unit, 158, 282, 285, 850
Synergistic Processing Element, 851
Synergistic Processor Element, 63
Synergistic Processor Unit, 63, 852
system, 29, 854
System Call interrupt, 255
system configurations, 162
System Error, 348
System Error interrupt, 257
system memory, 230, 391, 557, 854
System Reset Interrupt, 245
System Reset interrupt, 245
system storage, 41, 73, 854

T

T, 266, 277, 278, 279
table lookup, 89, 702
table lookup (tablewalk), 89
table walk, 89, 854
tablewalk, 89
tag group, 41, 456, 511, 524, 562, 563, 854
Tag Group ID, 72
tag ID, 72, 456
Tag-Group Dependencies, 512
Tag-Group Status Update Event, 486, 499
taken, 854
targets, 287
task, 42, 601, 854
TB, 332, 376
TBEN, 377
TBR, 376
TClassID, 854
TclassID, 455
TClassID0, 513
TClassID1, 513
TClassID2, 513
TE, 189, 192
text segment, 566, 854
Tg, 474
Thermal Management, 423
Thermal Management interrupt, 260
thermal overload, 427
Thermal Registers, 429
Thermal Sensor Interrupt registers, 429
Thermal Sensor Status registers, 429
Thermal-Management Control registers, 432
Thermal-Management Control Unit (TMCU), 426
Thermal-Management Stop Time registers, 432
Thermal-Management Throttle Point register, 432
Thermal-Management Throttle Scale register, 432
thrashing, 606
thread, 42, 855
thread library, 617
thread model, 34, 42, 841
threads, 50, 608
throughput, 53, 58, 296, 711, 712, 738, 855
ticks, 375
time base, 375, 855
Time Base Register (TB), 376
Time Base Register (TBR), 376
Time-Base Enable (TBEN), 377
time-base frequency, 375, 377
timebase_mode, 378
Timebase_setting, 378
Time-of-Day, 383
time-sharing, 328
TIS, 402, 855
TKM, 203, 212, 855
TLB, 47, 74, 760, 855
TLB hit, 87, 94, 110, 855
TLB invalidation, 733
TLB miss, 80, 87, 88, 92, 94
tlbie, 733, 855
TLBIE request, 116, 855
tlbiel, 733, 735
tlbsync, 554, 855
Tm, 474
TMCU, 426, 855
TOC, 396, 613, 855
TOE, 407, 855
TOE shadow, 409, 856
token, 203, 514, 856
Token Available Latches, 212
Token Management Unit, 855
Token Manager (TKM), 203, 212
Token Manager Interrupt, 273
Token-Available Latch, 218
tokens, 207
Tool Interface Standard (TIS), 402
touch, 137, 856
Trace Array, 439
trace array, 438
Trace interrupt, 256
traditional vector processor, 619
Transfer Class ID, 455, 513, 514
transfer element, 856
Transfer Size, 454
Translation Lookaside Buffer, 855
trap instruction, 55, 253, 856
triangle subdivision, 622
trip count, 633, 635, 646, 849, 856
truncation, 68
TS, 458, 856
U, 461

U, V, W

unaligned loads, 707
unaligned stores, 707
Unallocated RAG, 212, 214
unified cache, 37, 45, 856
unified register file, 67, 856
unordered, 193, 856
unpack, 661
unroll, 76, 284, 607, 610, 644, 666, 856
unrolled loops, 646
Unused Tokens, 218
UPC, 599, 609, 856
update, 52, 58, 72, 856
user mode, 301, 331, 856
user state, 301, 331
V, 86, 88, 169, 170, 171, 174, 266, 345, 781, 856
VA, 47, 856
valid, 86, 88, 106, 107, 145, 149, 170, 856
valid SIMDization, 642
vector, 55, 619, 857
vector data types, 774
vector literals, 60, 774
vector multimedia extensions, 45
vector operands, 55
vector token, 59, 744
vector types, 59, 744
Vector/SIMD Multimedia Extension, 857
Vector/SIMD Multimedia Extension instruction set, 55, 58, 737
vector-across form, 620, 823
vectorization, 57
vectorizing compilers, 637
VIQ, 754, 857
Virtual Address, 47, 81, 93, 856
virtual channels, 189
virtual memory, 79, 857
virtual mode, 857
Virtual Page Number, 87, 857
virtual vector, 644, 645, 663, 857
vload, 649
volatile register, 375, 487, 489, 496, 857
volatile variables, 687
vop, 649
vpack, 649
VPN, 87, 104, 114, 857
VR, 50
VRs, 50
VRSAVE, 50, 670
VSCR, 50, 670
vshiftpair, 649
vshiftstream, 649
VSID, 85, 93, 104, 857
vsplat, 649
vsplice, 649
vstore, 649
VSU, 46, 857
vunpack, 649
VXU, 46, 55, 239, 244, 256, 298, 299, 303, 325, 663, 738, 857
VXU Unavailable interrupt, 256
W, 88, 91, 114, 857
wait on external event, 466
Watchdog Timer, 499
WAW, 139, 753, 756, 857
WAW dependencies, 759
way, 92, 107, 857
weak ordering, 91, 857
weakly consistent storage order, 91, 551, 857
weakly ordered transfer model, 686
while loop construct, 635
WIMG bits, 91, 100, 857
word, 30, 858
workload, 327, 603, 617, 699, 858
WPC, 760, 858
WPC collisions, 760, 858
wrch, 442, 446, 858
write back, 717
write through, 858
Write With Clean, 858
Write With Flush, 858
Write With Kill, 858
Writeback, 144, 194, 533, 719, 771, 858
writeback, 575
writeback DMA command, 533
writeback flag, 858
Write-through, 46, 88, 91, 137, 552, 557
write-through, 552

X

XDR, 38, 215, 858
XER, 50
XIO, 858
XLAT, 135, 136, 150, 858
XU, 298, 717, 858

Z

zero, 68
zero-shift policy, 649, 650, 858

Revision Log

Revision Date	Version	Contents of Modification
April 19, 2006	1.0	