Cell Broadband Engine Architecture

Version 1.02

October 11, 2007

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

October 11, 2007
Version 1.02

Contents

List of Figures 11
List of Tables 13
Preface 15
Who Should Read This Manual 15
Document Organization 15
Related Publications 16
Conventions and Notation 17
References to Registers, Fields, and Bits 18
Endian Order 18
Revision Log 19
1. Introduction to Cell Broadband Engine Architecture 27
1.1 Organization of a CBEA-Compliant Processor 27
1.1.1 PowerPC Processor Element 29
1.1.2 Synergistic Processor Unit 29
1.1.3 Memory Flow Controller 29
1.1.4 Internal Interrupt Controller 30
1.2 Storage Types 30
1.2.1 Local Storage Addressing 30
1.2.2 Main Storage Addressing 31
1.2.3 Main Storage Attributes 31
1.3 Cache Replacement Management Facility 32
1.4 Instructions, Commands, and Facilities 32
1.5 Reserved Fields and Registers 32
1.6 Implementation-Dependent Fields and Registers 33
User Mode Environment 35
2. Overview 36
2.1 Instruction and Command Classes 36
2.1.1 Defined Class 36
2.1.2 Illegal Class 37
2.1.3 Reserved Class 37
2.2 Forms of Defined Instructions and Commands 37
2.2.1 Preferred Forms 37
2.2.2 Invalid Forms 38
2.2.3 Optional Forms 38
2.2.4 Optional Fields 38
2.3 Exceptions 38
2.4 SPU Events 39
3. Storage Models 41
3.1 Virtual Storage Model 41
3.2 SPU Local Storage Model 41
3.2.1 Local Storage Access 42
3.2.1.1 Mapping Requirements 42
3.2.1.2 Local Storage Access Exceptions 42

3.3 Single-Copy Atomicity 42
3.4 Cache Models 43
3.5 Memory Coherence 43
3.6 Storage Control Attributes 44
3.7 Shared Storage 45
4. PowerPC Processor Element 47
4.1 PowerPC Architecture, Book I and Book II Compatibility 47
4.1.1 Optional Features in PowerPC Architecture, Book I (Required for CBEA) 47
4.1.2 Incompatibilities with PowerPC Architecture, Book I 47
4.1.3 Optional Features in PowerPC Architecture, Book II (Required for CBEA) 48
4.1.4 Incompatibilities with PowerPC Architecture, Book II 48
4.1.5 Extensions to the PowerPC Architecture, Books I and II 48
5. Synergistic Processor Unit 49
6. Memory Flow Controller 51
6.1 MFC Facilities 52
7. MFC Commands 55
7.1 Command Classes 57
7.1.1 Defined Commands 57
7.1.2 Illegal Commands 60
7.1.3 Reserved Commands 61
7.2 Command Exceptions 61
7.3 MFC Command Parameters 62
7.4 List Commands and List Elements 63
7.5 Get Commands (Main Storage to Local Storage) 64
7.5.1 Get Command 64
7.5.2 Get with Fence or with Barrier Command 64
7.5.3 Get List Command 64
7.5.4 Get List with Fence or with Barrier Command 65
7.6 Put Commands (Local Storage to Main Storage) 65
7.6.1 Put Command 65
7.6.2 Put with Fence or with Barrier Command 66
7.6.3 Put List Command 66
7.6.4 Put List with Fence or with Barrier Command 66
7.6.5 Put Result (hint) Command 66
7.7 Storage Control Commands 67
7.7.1 SL1 Data Cache Range Touch Command 67
7.7.2 SL1 Data Cache Range Touch for Store Command 68
7.7.3 SL1 Data Cache Range Set to Zero Command 68
7.7.4 SL1 Data Cache Range Store Command 68
7.7.5 SL1 Data Cache Range Flush Command 69
7.8 MFC Atomic Update Commands 70
7.8.1 Get Lock Line and Reserve Command 70
7.8.2 Put Lock Line Conditional Command 71
7.8.3 Put Lock Line Unconditional Command 72
7.8.4 Put Queued Lock Line Unconditional Command 72
7.9 MFC Synchronization Commands 73
7.9.1 MFC Synchronize Command 75

7.9.2 MFC Enforce In Order Execution of I/O Command 76
7.9.3 Barrier Command 76
7.9.4 Send Signal Command 77
7.9.5 Send Signal with Fence or with Barrier Command 78

8. Problem-State Memory-Mapped Registers 79

8.1 MFC Proxy Command Parameter Registers 80
8.1.1 MFC Command Opcode Register 81
8.1.2 MFC Class ID Register 82
8.1.3 MFC Command Tag Register 83
8.1.4 MFC Transfer Size Register 84
8.1.5 MFC Local Storage Address Register 85
8.1.6 MFC Effective Address High Register 86
8.1.7 MFC Effective Address Low Register 87
8.2 MFC Proxy Command Issue Sequence 88
8.3 MFC Proxy Command Queue Status and Control Registers 89
8.3.1 MFC Command Status Register 90
8.3.2 MFC Queue Status Register 91
8.4 Proxy Tag-Group Completion Facility 92
8.4.1 Proxy Tag-Group Query Type Register 93
8.4.2 Proxy Tag-Group Query Mask Register 94
8.4.3 Proxy Tag-Group Status Register 95
8.5 SPU Control and Status Facilities 96
8.5.1 SPU Run Control Register 96
8.5.2 SPU Status Register 97
8.5.3 SPU Next Program Counter Register 99
8.6 Mailbox Facility 101
8.6.1 SPU Outbound Mailbox Register 102
8.6.2 SPU Inbound Mailbox Register 103
8.6.3 SPU Mailbox Status Register 104
8.7 SPU Signal Notification Facility 105
8.7.1 SPU Signal Notification 1 Register 106
8.7.2 SPU Signal Notification 2 Register 107
8.8 MFC Multisource Synchronization Facility 108
8.8.1 MFC Multisource Synchronization Register 109

9. Synergistic Processor Unit Channels 113

9.1 MFC SPU Command Parameter Channels 116
9.1.1 MFC Command Opcode Channel 117
9.1.2 MFC Class ID Channel 118
9.1.3 MFC Command Tag Identification Channel 119
9.1.4 MFC Transfer Size or List Size Channel 120
9.1.5 MFC Local Storage Address Channel 121
9.1.6 MFC Effective Address Low or List Address Channel 122
9.1.7 MFC Effective Address High Channel 124
9.2 MFC SPU Command Issue Sequence 125
9.3 MFC Tag-Group Status Channels 126
9.3.1 Determining the Status of Tag Groups 126
9.3.2 Determining Command Completion 128
9.3.3 MFC Write Tag-Group Query Mask Channel 129
9.3.4 MFC Read Tag-Group Query Mask Channel 131

9.3.5 MFC Write Tag Status Update Request Channel 132
9.3.6 MFC Read Tag-Group Status Channel 133
9.3.7 MFC Read List Stall-and-Notify Tag Status Channel 135
9.3.8 MFC Write List Stall-and-Notify Tag Acknowledgment Channel 136
9.4 MFC Read Atomic Command Status Channel 137
9.5 SPU Mailbox Channels 138
9.5.1 SPU Write Outbound Mailbox Channel 139
9.5.2 SPU Write Outbound Interrupt Mailbox Channel 140
9.5.3 SPU Read Inbound Mailbox Channel 141
9.6 SPU Signalling Channels 142
9.6.1 SPU Signal Notification 1 Channel 143
9.6.2 SPU Signal Notification 2 Channel 144
9.7 SPU Decrementer 145
9.7.1 SPU Write Decrementer Channel 145
9.7.2 SPU Read Decrementer Channel 146
9.8 SPU Read Machine Status Channel 147
9.9 SPU Interrupt-Related Channels 148
9.9.1 SPU Write State Save-and-Restore Channel 148
9.9.2 SPU Read State Save-and-Restore Channel 148
9.10 MFC Write Multisource Synchronization Request Channel 149
9.11 SPU Event Facility 150
9.11.1 SPU Read Event Status Channel 153
9.11.2 SPU Write Event Mask Channel 157
9.11.3 SPU Read Event Mask Channel 159
9.11.4 SPU Write Event Acknowledgment Channel 161
9.12 SPU Event Definitions 163
9.12.1 MFC Tag-Group Status Update Event 163
9.12.2 MFC List Command Stall-and-Notify Event 163
9.12.3 MFC SPU Command Queue Available Event 165
9.12.4 SPU Inbound Mailbox Available Event 166
9.12.5 SPU Decrementer Event 166
9.12.6 SPU Outbound Interrupt Mailbox Available Event 167
9.12.7 SPU Outbound Mailbox Available Event 168
9.12.8 SPU Signal Notification 2 Available Event 169
9.12.9 SPU Signal Notification 1 Available Event 170
9.12.10 Lock Line Reservation Lost Event 170
9.12.11 Privileged Attention Event 172
9.12.12 Multisource Synchronization Event 172

10. Storage Access Ordering 175
10.1 Order of Command Execution 177
10.2 Main Storage Domain Access Ordering 177
10.3 Local Storage Domain Access Ordering 179
10.4 Cross-Domain Storage Access Order 180
10.5 Cumulative Access Ordering 180
10.6 MFC Overlapped Accesses 180
10.7 Atomic Accesses 180
10.8 Store Combining 181
10.9 Storage Ordering of I/O Accesses 181

11. SPU Isolation Facility 183
11.1 SPU Isolation Facility Features 183
11.2 SPU Operating States 185

Privileged Mode Environment 187

12. Overview 189
12.1 Privileged Mode Facility Organization 189
12.1.1 SPE Privilege 1 Facilities 190
12.1.2 SPE Privilege 2 Facilities 193
12.1.3 PPE Privilege 1 Facilities 195

13. PowerPC Architecture, Book III Compatibility 197
13.1 Optional Features in PowerPC Architecture, Book III (Required for CBEA) 197
13.2 Incompatibilities with PowerPC Architecture, Book III 197
13.3 Extensions to the PowerPC Architecture 197

14. Storage Addressing 199
14.1 PPE Segment Lookaside Buffer Management 200
14.2 SPE Segment Lookaside Buffer Management 200
14.2.1 SLB Mapping 200
14.2.2 SLB Index Register 201
14.2.3 SLB Effective Segment ID Register 202
14.2.4 SLB Virtual Segment ID Register 203
14.2.5 SLB Invalidate Entry Register 205
14.2.6 SLB Invalidate All Register 206
14.3 Translation Lookaside Buffer Management 207
14.3.1 TLB Mapping 208
14.3.2 TLB Index Hint Register 209
14.3.3 TLB Index Register 210
14.3.4 TLB Virtual Page Number Register 211
14.3.5 TLB Real Page Number Register 212
14.3.6 TLB Invalidate Entry Register 214
14.3.7 TLB Invalidate All Register 216
14.4 Real-Mode Storage Control Facilities 217
14.4.1 PPE Real-Mode Storage Control Facility 217
14.4.2 MFC Real-Mode Address Boundary Register 218

15. MFC Privileged Facilities 221
15.1 MFC State Register One 221
15.2 MFC Logical Partition ID Register 223
15.3 MFC Storage Description Register 224
15.4 MFC Data Address Register 225
15.5 MFC Data Storage Interrupt Status Register 226
15.6 MFC Address Compare Control Register 227
15.7 MFC Local Storage Address Compare Facility 229
15.7.1 MFC Local Storage Address Compare Register 229
15.7.2 MFC Local Storage Compare Result Register 230
15.8 MFC Command Error Register 231
15.9 MFC Data Storage Interrupt Pointer Register 232

15.10 MFC Control Register 233
15.11 MFC Atomic Flush Register 236
15.12 SPU Outbound Interrupt Mailbox Register 237

16. SPU Privileged Facilities 239
16.1 SPU Privileged Control Register 239
16.2 SPU Local Storage Limit Register 241
16.3 SPU Channel Access Facility 242
16.3.1 SPU Channel Index Register 242
16.3.2 SPU Channel Data Register 243
16.3.3 SPU Channel Count Register 244
16.4 SPU Configuration Register 245

17. SPE Context Save and Restore 247

18. PPE Address Range Facility 249
18.1 Range Start Register 251
18.2 Range Mask Register 252
18.3 Class ID Register 253

19. Cache Replacement Management Facility 255
19.1 Replacement Management Table Example 255
19.2 RMT Index Generation Example 256
19.2.1 RMT Index Register 257
19.2.2 RMT Data Register 258

20. Resource Allocation Management 259

21. Interrupt Facilities 261
21.1 Interrupt Classification 261
21.2 Interrupt Presentation 262
21.3 Internal Interrupt Controller Registers 263
21.3.1 Interrupt Pending Port Registers 263
21.3.2 Interrupt Generation Port Register 267
21.3.3 Interrupt Current Priority Level Register 268
21.4 SPU and MFC External Interrupt Definitions 269
21.5 SPU and MFC Interrupt Generation Process 270
21.5.1 Class 0 Interrupts 270
21.5.2 Class 1 Interrupts 272
21.5.3 Class 2 Interrupts 274
21.6 MFC Interrupt Mask Registers 276
21.6.1 Class 0 Interrupt Mask Register 276
21.6.2 Class 1 Interrupt Mask Register 277
21.6.3 Class 2 Interrupt Mask Register 278
21.7 MFC Interrupt Status Registers 279
21.7.1 Class 0 Interrupt Status Register 280
21.7.2 Class 1 Interrupt Status Register 281
21.7.3 Class 2 Interrupt Status Register 282
21.8 Interrupt Routing Register 283

22. Power Management 285

23. Version Control 287
23.1 CBEA-Compliant Processor Version Register 287
23.2 PPE Processor Version Register 288
23.3 SPU Version Register 289
23.4 MFC Version Register 290
23.5 SPU Identification Register 291

Appendix A. Memory Maps 293
A.1 SPE Problem State Memory Map 296
A.2 SPE Privilege 1 Memory Map 298
A.3 SPE Privilege 2 Memory Map 301
A.4 PPE Privilege 1 Memory Map 302
A.5 Internal Interrupt Controller Memory Map 303

Appendix B. SPU Channel Map 305

Appendix C. CBEA-Specific PPE Special Purpose Registers 309

Appendix D. Defined Commands 311

Appendix E. Extensions to the PowerPC Architecture 315
E.1 Software Management of TLBs (optional) 315
E.2 Mediated External Exception Extension (optional) 316
E.2.1 Using the Mediated External Exception Extension 317
E.3 Multiple Concurrent Large Pages (optional) 318
E.4 Defined Behavior for Inaccessible SPRs 319
E.5 Vector/SIMD Multimedia Extension (optional) 319

Appendix F. Examples of Access Ordering 321

Glossary 329

Index 341

List of Figures

Figure 1-1. CBEA-Compliant Processor System 28
Figure 6-1. Typical MFC Block Diagram 51
Figure 9-1. Logical Representation of SPU Event Support 151
Figure 10-1. Storage Domains in a CBEA-Compliant Processor 176
Figure 11-1. SPU State Transitions 186
Figure 14-1. Real-Mode Storage Boundary (showing instruction fetches and data fetches) 217
Figure 14-2. Real-Mode Storage Boundary (showing all DMA transfers) 218
Figure 18-1. Generation of Class ID from the Address Range Registers 249
Figure 19-1. RMT Index Generation 257
Figure 21-1. Interrupt Presentation 262
Figure 21-2. MFC Class 0 Interrupt Generation Process 271
Figure 21-3. MFC Class 1 Interrupt Generation Process 273
Figure 21-4. MFC Class 2 Interrupt Generation Process 275

List of Tables

Table 1-1. Sizes of Main Storage Address Spaces 31
Table 7-1. Parameter Mnemonics 56
Table 7-2. Data Transfer or MFC DMA Commands 58
Table 7-3. SL1 Storage Control Commands 59
Table 7-4. MFC Synchronization Commands 60
Table 7-5. MFC Atomic Commands 60
Table 7-6. Command Errors and Alignment Errors 61
Table 8-1. SPE Problem-State Memory Map 79
Table 9-1. SPU Channel Instructions 113
Table 9-2. SPU Channel Map 114
Table 12-1. SPE Privilege 1 Memory Map 190
Table 12-2. SPE Privilege 2 Memory Map 193
Table 12-3. PPE Privilege 1 Memory Map 195
Table 19-1. Typical RMT Entry for an 8-Way Set Associative Cache 255
Table 21-1. Internal Interrupt Controller Memory Map 263
Table 21-2. SPU and MFC Interrupt Class Definitions 269
Table 21-3. SPU and MFC External Interrupt Definitions 269
Table 22-1. Power Management States 285
Table A-1. CBEA-Compliant Processor Memory Map 294
Table A-2. SPE Problem State Memory Map 296
Table A-3. SPE Privilege 1 Memory Map 298
Table A-4. SPE Privilege 2 Memory Map 301
Table A-5. PPE Privilege 1 Memory Map 302
Table A-6. Internal Interrupt Controller Memory Map 303
Table B-1. SPU Channel Map 305
Table C-1. PPE Special Purpose Register Map 309
Table D-1. Parameter Mnemonics 311
Table D-2. Data Transfer or MFC DMA Commands 312
Table D-3. SL1 Storage Control Commands 313
Table D-4. MFC Synchronization Commands 314
Table D-5. MFC Atomic Commands 314

Preface

This document defines the Cell Broadband Engine™ Architecture (CBEA). The information contained in this document allows various CBEA-compliant processor implementations to meet the needs of a wide variety of systems and applications. Compatibility with the CBEA allows applications and privileged software to migrate from one implementation to another with minor changes.

For a specific implementation of the CBEA, see the specific implementation documentation.

Who Should Read This Manual

This manual is intended for designers who plan to develop products that use the CBEA. Readers of this manual should be familiar with the documents listed in *Related Publications* on page 16. In addition, individual implementations of the CBEA should have their own documentation for that implementation.

Document Organization

The CBEA document is divided into two environments: the user mode environment (UME) and the privileged mode environment (PME). In general, the UME describes the commands and facilities available to an application. The PME describes the facilities available to an operating system or to hypervisor code. Together, these two environments define the CBEA. Implementation details and compliance with the architecture for a specific implementation are provided separately.

Document Division	Description
Preface	Describes this document, related documents, the intended audience, and other general information.
Revision Log	Lists all significant changes made to the document since its initial release.
Introduction	Provides a high-level overview of the Cell Broadband Engine Architecture (CBEA).
User Mode Environment	Defines the base instruction set, command set, storage models, and facilities available to an application programmer, as well as compatibility with the PowerPC Architecture™. The following sections are included: • Overview • Storage Models • PowerPC Processor Element • Synergistic Processor Unit • Memory Flow Controller • MFC Commands • Problem-State Memory-Mapped Registers • Synergistic Processor Unit Channels • Storage Access Ordering • SPU Isolation Facility

Document Division	Description
Privileged Mode Environment	Describes the additional instructions and facilities, beyond those defined in user mode environment, that are provided by the CBEA. This division covers instructions and facilities, not available to the application programmer, that affect storage control, interrupts, and timing facilities. The following sections are included: • Overview • PowerPC Architecture, Book III Compatibility • Storage Addressing • MFC Privileged Facilities • SPU Privileged Facilities • SPE Context Save and Restore • PPE Address Range Facility • Cache Replacement Management Facility • Resource Allocation Management • Interrupt Facilities • Power Management • Version Control
Appendixes	• *Appendix A* maps all the registers defined in the Cell Broadband Engine Architecture to the real address space. • *Appendix B* maps all the channels defined by the Cell Broadband Engine Architecture in the real address space. • *Appendix C* lists the special-purpose registers (SPRs) required by the Cell Broadband Engine Architecture. • *Appendix D* lists the memory flow controller (MFC) commands. • *Appendix E* describes instructions and facilities that are extensions to the PowerPC Architecture. • *Appendix F* provides examples of access ordering.
Glossary	Defines terms and acronyms used in this document

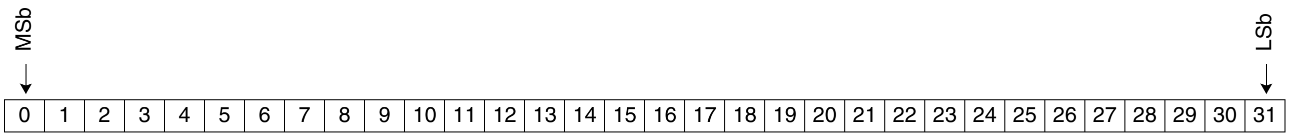
Related Publications

A list of documents related to the *CBEA* follows.

Title	Version	Date
PowerPC User Instruction Set Architecture, Book I	2.02	January 2005
PowerPC Virtual Environment Architecture, Book II	2.02	January 2005
PowerPC Operating Environment Architecture, Book III	2.02	January 2005
PowerPC Microprocessor Family: The Programming Environments Manual for 64-Bit Microprocessors	3.0	July 2005
PowerPC Microprocessor Family: Vector/SIMD Multimedia Extension Technology Programming Environments Manual	2.07c	October 2006
Synergistic Processor Unit Instruction Set Architecture	1.2	January 2007

Conventions and Notation

This document uses standard IBM notation, 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 bit, and bit 31 is the least-significant 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’.

This document uses the following software documentation conventions:

1. Command or instruction names are written in **bold** type. For example: **put**, **get, rdch, wrch,** and **rchcnt**.
2. 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]***.

This document uses the following symbols:

&	bitwise AND
|	bitwise OR
%	modulus
=	equal to
! =	not equal to
≥	greater than or equal to
≤	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 zeros

References to Registers, Fields, and Bits

Registers are referred to by their full name or by their short name (also called the register mnemonic). Fields are referred to by their field name or by their bit position. The following table describes how registers, fields, and bit ranges are referred to in this document and provides examples.

Type of Reference	Format	Example
Reference to a specific register and a specific field using the register short name and the field name	Register_Short_Name[Field_Name]	MSR[R]
Reference to a field using the field name	[Field_Name]	[R]
Reference to a specific register and to multiple fields using the register short name and the field names	Register_Short_Name[Field_Name1, Field_Name2]	MSR[FE0, FE1]
Reference to a specific register and to multiple fields using the register short name and the bit positions.	Register_Short_Name[Bit_Number, Bit_Number]	MSR[52, 55]
Reference to a specific register and to a field using the register short name and the bit position or the bit range.	Register_Short_Name[Bit_Number]	MSR[52]
	Register_Short_Name[Starting_Bit_Number:Ending_Bit_Number]	MSR[39:44]

A field name followed by an equal sign (=) and a value indicates the value for that field.	Register_Short_Name[Field_Name]=*n*[1]	MSR[FE0]='1'
		MSR[FE]=x'1'

	Register_Short_Name[Bit_Number]=*n*[1]	MSR[52]='0'
		MSR[52]=x'0'

	Register_Short_Name[Starting_Bit_Number:Ending_Bit_Number]=*n*[1]	MSR[39:43]='10010' MSR[39:43]=x'11'

1. Where *n* is the binary or hexadecimal value for the field or bits specified in the brackets.

Endian Order

The PowerPC Architecture supports both big-endian and little-endian byte-ordering modes. *Book I* of the *PowerPC Architecture* describes these modes. For more information about big-endian and little-endian byte ordering, see the following sections in that book:

- *Big Endian Mapping*
- *Little Endian Mapping*

The CBEA supports only big-endian byte ordering. Therefore, PowerPC® Processor Elements (PPEs) in a CBEA-compliant implementation are *not* required to support the little-endian byte-ordering mode as defined in the PowerPC Architecture. Synergistic processor units (SPUs) do *not* implement the optional little-endian byte-ordering mode. The PowerPC Processor Element (PPE) structure mapping is identical to that used for the SPUs in a CBEA-compliant system.

Because the CBEA supports only big-endian byte ordering, the memory flow controller (MFC) direct memory access (DMA) command and control registers do *not* implement the optional little-endian byte-ordering mode. The DMA data transfers themselves are simply byte moves, without regard to the numerical significance of any byte. Thus, the big-endian or little-endian issue becomes irrelevant to the actual movement of a block of data. The byte-order mapping only becomes significant when data is fetched or interpreted, for example by a processor, or by an MFC.

Revision Log

Each release of this document supersedes all previously released versions. The revision log lists all significant changes made to the document since its initial release. In the rest of the document, change bars in the margin indicate that the adjacent text was significantly modified from the previous release of this document.

Revision Date	Version	Contents of Modification
October 11, 2007	1.02	Second revision (1.02). • Changed “physical address space” to “real address space” (see *Document Organization* on page 15, *Appendix B SPU Channel Map* on page 305, and *Glossary* on page 329). • Added Appendix E to the document organization table (see *Document Organization* on page 15). • Updated the list of documents related to the CBEA (see *Related Publications* on page 16). • Corrected a sentence indicating that the vector/SIMD multimedia extension unit is required in the PPE of a CBEA-compliant system; it is optional (see *Section 1.1.1 PowerPC Processor Element* on page 29). • Removed a redundant sentence from a programming note (see *Section 1.5 Reserved Fields and Registers* on page 32). • Indicated that the preferred forms of the PPE instructions are defined in the *PowerPC Architecture* (see *Section 2.2.1 Preferred Forms* on page 37). • Listed the optional SPU instructions (see *Section 2.2.3 Optional Forms* on page 38). • Changed “Synergistic Processor Element (SPU)” to “synergistic processor unit (SPU)” (see *Section 5 Synergistic Processor Unit* on page 49). • Changed “Synergistic Processor Element (SPE)” to “synergistic processor unit (SPU)” and “SPE” to “SPU” (see *Section 7 MFC Commands* on page 55). • Rewrote the descriptions of the “f” and “b” command modifiers (see *Section 7 MFC Commands* on page 55). • Indicated that 0 is a valid transfer size (see *Table 7-6 Command Errors and Alignment Errors* on page 61). • Reworded the description of get commands (see *Section 7.5 Get Commands (Main Storage to Local Storage)* on page 64). • Reworded the description of the get list (**getl**) command, indicating that the local storage address (LSA) must start on a 16-byte boundary *unless* (instead of *if*) the transfer size of the first list element is less than 16 bytes (see *Section 7.5.3 Get List Command* on page 64). • Reworded the description of the put list (**putl**) command, indicating that the local storage address (LSA) must start on a 16-byte boundary *unless* (instead of *if*) the transfer size of the first list element is less than 16 bytes (see *Section 7.6.3 Put List Command* on page 66). • Reworded the description of the get lock line and reserve (**getllar**) command (see *Section 7.8.1 Get Lock Line and Reserve Command* on page 70). • Reworded an implementation note (see *Section 7.9 MFC Synchronization Commands* on page 73). • Clarified the order in which commands are performed when a barrier modifier is used (see *Section 7.9 MFC Synchronization Commands* on page 75). • Reworded the description of the send signal command (**sndsig**) (see *Section 7.9.4 Send Signal Command* on page 77). • Changed “MFC Class ID Register” to “MFC class ID” (see *Section 8.1.2 MFC Class ID Register* on page 82). • Indicated that 0 is a valid transfer size and corrected a cross reference (see *Section 8.1.4 MFC Transfer Size Register* on page 84). • Added footnotes indicating that the effective address can be written using either one 64-bit store or two 32-bit stores. However, 64-bit access to an address range that includes a 32-bit MMIO register is generally not allowed unless explicitly specified (see *Section 8.1.6 MFC Effective Address High Register* on page 86 and *Section 8.1.7 MFC Effective Address Low Register* on page 87). • Expanded a footnote to indicate that 64-bit access to an address range that includes a 32-bit MMIO register is generally not allowed unless explicitly specified (see *Section 8.2 MFC Proxy Command Issue Sequence* on page 88). • Corrected a cross reference (see *Section 8.3 MFC Proxy Command Queue Status and Control Registers* on page 89). • Revised the description of the MFC Command Status Register (see *Section 8.3.1 MFC Command Status Register* on page 90)

Revision Date	Version	Contents of Modification
October 11, 2007 (continued)	1.02	• Removed an incorrect statement about the MFC proxy command and expanded the description of the proxy tag-group completion facility (see *Section 8.4 Proxy Tag-Group Completion Facility* on page 92). • Added a note indicating that a store operation to the SPU_RunCntl Register is a context synchronizing operation (see *Section 8.5.1 SPU Run Control Register* on page 96). • Explained that the contents of the signal notification registers are reset to zero when the SPU reads the value of the corresponding channels (see *Section 8.7.1 SPU Signal Notification 1 Register* on page 106 and *Section 8.7.2 SPU Signal Notification 2 Register* on page 107). • Reworded the description of the MFC Multisource Synchronization Register to indicate that the MFC Write Multisource Synchronization Request Channel is also part of the MFC multisource synchronization facility. Where the term “MFC multisource synchronization facility” was incorrectly used, changed it to “MFC Multisource Synchronization Register” (see *Section 8.8.1 MFC Multisource Synchronization Register* on page 109). • Reworded the description of the MFC Command Opcode Channel (see *Section 9.1.1 MFC Command Opcode Channel* on page 117). • Changed “proxy” to “SPU” and “MFC Class ID Channel” to “MFC class ID” (see *Section 9.1.2 MFC Class ID Channel* on page 118). • Indicated the type of interrupt generated if the MFC command tag identification parameter is not valid (see *Section 9.1.3 MFC Command Tag Identification Channel* on page 119). • Corrected a cross reference and changed the programming note associated with the MFC Transfer Size or List Size Channel (see *Section 9.1.4 MFC Transfer Size or List Size Channel* on page 120). • Changed “MFC State Register” to “MFC State Register One” and “MFC_SR” to “MFC_SR1” (see *Section 9.1.6 MFC Effective Address Low or List Address Channel* on page 122 and *Section 9.1.7 MFC Effective Address High Channel* on page 124). • Expanded the description of the MFC tag-group status channels (see *Section 9.3 MFC Tag-Group Status Channels* on page 126). • Modified the procedure for using the SPU event facility (see *Section 9.3.2 Determining Command Completion* on page 129). • Completed a sentence (see *Section 9.3.5 MFC Write Tag Status Update Request Channel* on page 132). • Expanded the description of the MFC Read Tag-Group Status Channel (see *Section 9.3.6 MFC Read Tag-Group Status Channel* on page 133). • Provided a less restrictive description of successive decrementer count reads (see *Section 9.7.2 SPU Read Decrementer Channel* on page 146). • Added a programming note explaining how to avoid waiting on an event indefinitely when using the SPU event facility (see *Section 9.11 SPU Event Facility* on page 150). • Included a reference to *Figure 9-1* in the implementation note associated with the SPU Read Event Status Channel and corrected incorrect cross references (see *Section 9.11.1 SPU Read Event Status Channel* on page 156). • Changed “the count is set to 1” to “the count is incremented” in the description of the MFC SPU command queue available event (see *Section 9.12.3 MFC SPU Command Queue Available Event* on page 165). • Reworded the description of the lock line reservation lost event, changing “cache line” to “reservation granule.” Added a programming note explaining how to avoid or handle a situation where a lock line reservation lost event is not presented (see *Section 9.12.10 Lock Line Reservation Lost Event* on page 170). • Corrected a bit name (see *Section 9.12.11 Privileged Attention Event* on page 172). • Expanded the description of storage access ordering (see *Section 10 Storage Access Ordering* on page 175). • Specified that “A” is a set of *main* storage accesses in the example illustrating PPU execution of a **sync** or **eieio** instruction (see *Section 10.2 Main Storage Domain Access Ordering* on page 178). • Changed the description of MFC overlapped accesses to be consistent with *Section 3.2.1.2 Local Storage Access Exceptions* on page 42 and changed “address” to “access” (see *Section 10.6 MFC Overlapped Accesses* on page 180).

Revision Date	Version	Contents of Modification
October 11, 2007 (continued)	1.02	• Changed “hypervisor mode (HV = ‘1’) and problem mode (PR = ‘0’)” to “hypervisor mode (HV = ‘1’ and PR = ‘0’)” (see *Section 12.1 Privileged Mode Facility Organization* on page 189). • Removed a phrase indicating that certain extensions to the PowerPC Architecture are required in a CBEA-compliant system (see *Section 13.3 Extensions to the PowerPC Architecture* on page 197). • Added footnotes indicating that 64-bit access to an address range that includes a 32-bit MMIO register is generally not allowed unless explicitly specified (see *Section 14.2.3 SLB Effective Segment ID Register* on page 202, *Section 14.2.4 SLB Virtual Segment ID Register* on page 203, *Section 14.2.5 SLB Invalidate Entry Register* on page 205, and *Section 14.2.6 SLB Invalidate All Register* on page 206). • Rewrote an implementation note (see *Section 14.2.6 SLB Invalidate All Register* on page 206). • Changed the address offset of the TLB Virtual Page Number Register from x‘3B5’ to x‘3B4’; Changed the address offset of the TLB Real Page Number Register from x‘3B4’ to x‘3B5’ (see *Section 14.3.4 TLB Virtual Page Number Register* on page 211, *Section 14.3.5 TLB Real Page Number Register* on page 212, and *Table C-1 PPE Special Purpose Register Map* on page 309). • Corrected the description of TLB_RPN[LP] (see *Section 14.3.5 TLB Real Page Number Register* on page 212). • Corrected the TLB Invalidate Entry Register (see *Section 14.3.6 TLB Invalidate Entry Register* on page 214). • Corrected the size of the boundary between storage that is considered well behaved and storage that is not (see *Section 14.4.1 PPE Real-Mode Storage Control Facility* on page 217 and *Section 14.4.2 MFC Real-Mode Address Boundary Register* on page 218). • Corrected the base address offset of MFC_RMAB (see *Section 14.4.2 MFC Real-Mode Address Boundary Register* on page 218). • Changed “Local Storage Read bit (Lp)” to “Local Storage Put bit (Lp)” and provided names for the MFC_ACCR bits; revised the description of the Lp bit; removed an incorrect cross reference (see *Section 15.6 MFC Address Compare Control Register* on page 227). • Changed “PTE address compare” to “effective address compare” in several places (see *Section 15.6 MFC Address Compare Control Register* on page 227 and *Section 15.10 MFC Control Register* on page 233). • Specified the order of precedence in the equations associated with the MFC_LSACR (see *Section 15.7.1 MFC Local Storage Address Compare Register* on page 229). • Changed “SPU_Trapped interrupt” to “SPU halted interrupt.” Indicated that SPU_Status[IS] equals ‘1’ when the SPU is operating in isolation mode. Added a note to SPU_PrivCntl[A] (see *Section 16.1 SPU Privileged Control Register* on page 239). • Corrected several cross references (see *Section 19.2.1 RMT Index Register* on page 257 and *Section 19.2.2 RMT Data Register* on page 258). • Changed “thread number” to “PPU thread number” (see *Section 21.3 Internal Interrupt Controller Registers* on page 263). • Changed a “less than” sign to a “less than or equal to” sign (see *Table 21-1 Internal Interrupt Controller Memory Map* on page 263). • Indicated that the H bit in the SPU Status Register is set to ‘1’ and the H bit in the Class 2 Interrupt Status Register is set to ‘1’ if a single instruction step completes (see *Section 21.5.3 Class 2 Interrupts* on page 274). • Indicated that a pulse condition sets MFC class 1 interrupt status bits 60 and 61 and provided cross references to figures illustrating the interrupt generation process (see *Section 21.7 MFC Interrupt Status Registers* on page 279). • Indicated that read access to the Processor Version Register “is” (rather than “should be”) privileged (see *Section 23.1 CBEA-Compliant Processor Version Register* on page 287). • Changed “processors” to “SPUs” (see *Section 23.3 SPU Version Register* on page 289). • Changed “processors” to “MFCs” (see *Section 23.4 MFC Version Register* on page 290). • Corrected the list of memory map sections (see *Appendix A Memory Maps* on page 293). • Made several minor changes to *Table A-1 CBEA-Compliant Processor Memory Map* on page 294. • Corrected the address offset of the reserved portion of the control and configuration area (see *Table A-3 SPE Privilege 1 Memory Map* on page 298).

Revision Date	Version	Contents of Modification
October 11, 2007 (continued)	1.02	• Corrected the address offset of the reserved portion of the RMT area (see *Table A-5 PPE Privilege 1 Memory Map* on page 302). • Changed "thread number" to "PPU thread number" (see *Section A.5 Internal Interrupt Controller Memory Map* on page 303). • Changed a column heading in *Table C-1 PPE Special Purpose Register Map* on page 309. • Revised *Appendix E.2 Mediated External Exception Extension (optional)* on page 316. • Updated the examples of access ordering to reflect the ordering rules (see *Appendix F Examples of Access Ordering* on page 321). • Explained when the MFC local storage address (LSA) access is complete and corrected several cross references (see *Example 7* on page 323 and *Example 8* on page 323). • Changed **put** to **get** in step 3 of MFC (1) (see *Example 14* on page 325). • Added the following definition: barrier; modified the following definitions: DMA queue, EIB, error correction code, fence, vector/SIMD multimedia extension, virtual address (see *Glossary* on page 329). • Changed SPU_IDR to SPU_ID throughout to be consistent with the *Cell Broadband Engine Registers* document. • Made other minor changes for clarity and consistency.
October 3, 2006	1.01	First revision (1.01). • Reworded the description of the synergistic processor unit (see *Section 1.1.2* on page 29). • Reworded the description of the memory flow controller (see *Section 1.1.3* on page 29). • Reworded the description of local storage addressing (see *Section 1.2.1* on page 30). • Added optional forms to the list of forms available for defined instructions or commands (see *Section 2.1.1* on page 36). • Reworded the description of the illegal class of instructions and commands (see *Section 2.1.2* on page 37). • Indicated that there are no preferred forms of the defined instructions and commands (see *Section 2.2.1* on page 37). • Corrected a reference in *Section 2.2.2 Invalid Forms* on page 38. • Revised the list of exceptions caused directly by the execution of an instruction (see *Section 2.3* on page 38). • Added an item to the list of accesses that are always atomic (see *Section 3.3 Single-Copy Atomicity* on page 42). • Removed an inaccurate note in *Section 3.4 Cache Models* on page 43. • Removed the phrase "with the MFC" from the description of the PowerPC Processor Element (PPE) (see *Section 4* on page 47). • Removed vector/SIMD multimedia extension from the list of optional *PowerPC Architecture, Book I* instructions that are required by the CBEA. Expanded the description of the **fres** and **frsqte** instructions (see *Section 4.1.1 Optional Features in PowerPC Architecture, Book I (Required for CBEA)* on page 47). • Revised the description of the optional version of the **dcbt** instruction (see *Section 4.1.3 Optional Features in PowerPC Architecture, Book II (Required for CBEA)* on page 48). • Moved the list of requests for extensions to the PowerPC Architecture from Appendix E to *Section 4.1.5* on page 48 and to *Section 13.3* on page 197. • Reworded the description of memory flow controller (MFC) commands (see *Section 7* on page 55). • Included a loop in the code example illustrating a list command (see the notes after *Table 7-1* on page 56). • Reworded the description of defined commands (see *Section 7.1.1* on page 57). • Reworded the description of the **mfceieio** command (see *Table 7-4 MFC Synchronization Commands* on page 60). • Reworded the description of illegal commands (see *Section 7.1.2* on page 60). • Corrected the range of the MFC command opcode parameter for reserved commands (see *Section 7.1.3 Reserved Commands* on page 61). • Added a table summarizing the command and alignment errors that the CBEA identifies (see *Table 7-6* on page 61).

Revision Date	Version	Contents of Modification
October 3, 2006 (continued)	1.01	• Reworded the description of MFC command parameters. Included the missing list address (LA) parameter. Added cross references to the MFC SPU Command Parameter Channels. (See *Section 7.3 MFC Command Parameters* on page 62.) • Reworded the description of list commands and list elements (see *Section 7.4* on page 63). • Reworded the description of the get list command (see *Section 7.5.3* on page 64). • Reworded the description of put commands (see *Section 7.6* on page 65). • Reworded the descriptions of storage control commands and added implementation notes (see *Section 7.7* on page 67). • Revised the description of MFC atomic update commands, indicating that the SPE and PPE must have the same granule size (see *Section 7.8 MFC Atomic Update Commands* on page 70). • Revised the description of the **getllar** command. Added a programming note that explains the need to avoid looping on **getllar** commands (see *Section 7.8.1 Get Lock Line and Reserve Command* on page 70). • Expanded the description of the **putllc** command (see *Section 7.8.2 Put Lock Line Conditional Command* on page 71). • Revised the description of MFC synchronization commands (see *Section 7.9 MFC Synchronization Commands* on page 73). • Corrected the description of the **mfcsync** command (see *Section 7.9.1* on page 75). • Corrected the description of the **mfceieio** command (see *Section 7.9.2* on page 76). • Added an introduction to *Table 8-1 SPE Problem-State Memory Map* on page 79. • Changed the title of *Section 8.1* on page 80 from MFC Command Parameter Registers to MFC Proxy Command Parameter Registers. • Added a note to the description of the MFC Command Opcode Register (see *Section 8.1.1* on page 81). • Revised the description of the MFC Class ID Register (see *Section 8.1.2* on page 82). • Clarified the note associated with the MFC Command Tag Register (see *Section 8.1.3* on page 83). • Changed the programming note associated with the MFC Transfer Size Register (see *Section 8.1.4* on page 84). • Changed the programming note associated with the MFC Local Storage Address Register (see *Section 8.1.5* on page 85). • Changed the description of the MFC Effective Address High Register (see *Section 8.1.6* on page 86). • Changed the description of the MFC Effective Address Low Register and the programming note associated with it (see *Section 8.1.7* on page 87). • Added a footnote to the MFC proxy command issue sequence and revised the implementation note associated with the sequence (see *Section 8.2* on page 88). • Changed the description of the proxy tag-group completion facility. Changed the value written to the Proxy Tag-Group Query Type Register to request a tag-group query from ‘11’ to ‘10’ (see *Section 8.4* on page 92). • Changed SPU_NPC[30] from not implemented (NI) to Reserved (see *Section 8.5.3 SPU Next Program Counter Register* on page 99). • Reworded the description of the SPU signal notification facility (see *Section 8.7* on page 105). • Revised the description of the MFC multisource synchronization facility (see *Section 8.8* on page 108). • Reworded the description of the MFC Multisource Synchronization Register and the associated examples (see *Section 8.8.1* on page 109). • Reworded the general description of SPU channels (see *Section 9* on page 113). • Reworded the general description of MFC command parameter channels (see *Section 9.1* on page 116). • Indicated that the MFC Command Opcode Channel is blocking (see *Section 9.1 MFC SPU Command Parameter Channels* on page 116 and *Section 9.2 MFC SPU Command Issue Sequence* on page 125). • Revised the description of the MFC Class ID Channel (see *Section 9.1.2* on page 118).

Revision Date	Version	Contents of Modification
October 3, 2006 (continued)	1.01	• Added 16 bits to the MFC Command Tag Identification Channel (see *Section 9.1.3* on page 119) and to the MFC Transfer Size or List Size Channel (see *Section 9.1.4* on page 120). • Expanded the programming note associated with the MFC Local Storage Address Channel (see *Section 9.1.5* on page 121). • Reworded the description of the MFC Effective Address Low or List Address Channel (see *Section 9.1.6* on page 122). • Reworded the description of the MFC Effective Address High Channel (see *Section 9.1.7* on page 124). • Reworded the description of the MFC SPU command issue sequence (see *Section 9.2* on page 125). • Reworded the description of the MFC tag-group status channels (see *Section 9.3* on page 126). • Modified step 4 of the basic procedure for waiting for a list command to reach a list element with the stall-and-notify flag set (see *Section 9.3.2* on page 128). • Reworded the description of the MFC Write Tag-Group Query Mask Channel (see *Section 9.3.3* on page 129). • Reworded the description of the MFC Read Tag-Group Query Mask Channel (see *Section 9.3.4* on page 131). • Reworded the description of the MFC Write Tag Status Update Request Channel (see *Section 9.3.5* on page 132). • Reworded the description of the MFC Read List Stall-and-Notify Tag Status Channel and revised the associated implementation notes (see *Section 9.3.7* on page 135). • Revised the description of the SPU Read Machine Status Channel (see *Section 9.8* on page 147). • Changed SPU_RdEventStat[27] to indicate that the SPU inbound mailbox event is triggered when the SPU Read Inbound Mailbox Channel transitions from a zero to a nonzero value. Added the lock line reservation lost event to the list of events hardware detects (see *Section 9.11.1* on page 153). • Reworded step 6 of the procedure to handle an MFC list command stall-and-notify event (see *Section 9.12.2* on page 163). • Revised the description of a multisource synchronization event (see *Section 9.12.12* on page 172). • Redefined “main storage” (see *Section 10.1* on page 177). • Reworded the description of main storage domain access ordering (see *Section 10.2* on page 177). • Added *Section 10.9 Storage Ordering of I/O Accesses* on page 181. • Indicated that a load function must be initiated to change an SPU from an SPU nonisolated execution environment to an SPU isolated execution environment (see *Section 11.1* on page 183). • Reworded the general description of the privileged mode facility organization (see *Section 12.1* on page 189). • Corrected the offset of the reserved section of the Synergistic Processor Element (SPE) privilege 1 control and configuration area (see *Table 12-1* on page 190). • Added *Section 13.3 Extensions to the PowerPC Architecture* on page 197. • Reworded the description of storage addressing (see *Section 14* on page 199). • Integrated a subsection called “SLB Management” into the preceding paragraph (see *Section 14.2* on page 200). • Changed the range of SLB_VSID[LP] from 57-59 to 58 -59. Corrected the descriptions of the SLB_VSID[LP] and SLB_VSID[L] fields. (See *Section 14.2.4* on page 203.) • Corrected the description of the SLB Invalidate All Register to indicate that a write to the least-significant word causes *all* entries, not a single entry, to be invalidated (see *Section 14.2.6* on page 206). • Revised the description of TLB management and added a programming note and an implementation note (see *Section 14.3* on page 207). • Reworded the description of the TLB Virtual Page Number Register and the associated note (see *Section 14.3.4* on page 211). • Corrected the meaning of the first value in TLB_RPN[LP] (see *Section 14.3.5* on page 212).

Revision Date	Version	Contents of Modification
October 3, 2006 (continued)	1.01	• Corrected the meanings of the values in TLB_Invalidate_Entry[LP] (see *Section 14.3.6* on page 214). • Revised the programming note associated with MFC_SR1[D] (see *Section 15.1* on page 221). • Indicated that the MFC command queue status changes to MFC command queue operation suspended when all outstanding DMA transfers are complete (see *Section 15.10* on page 233). • Corrected the description of the SPU Channel Data Register to indicate that the external event mask has no effect on the data read when using the SPU channel access facility, not when using a read channel (**rdch**) instruction (see *Section 16.3.2* on page 243). • Corrected the note associated with the SPU Channel Count Register to indicate that the channel count for the MFC Command Opcode Channel must be initialized to an implementation-dependent maximum value, not to 16 (see *Section 16.3.3* on page 244). • Reworded the description of SPE Context Save and Restore (see *Section 17* on page 247). • Reworded the general description of the PPE address range facility (see *Section 18* on page 249). • Removed implementation-specific information from the description of resource allocation management (see *Section 20* on page 259). • Reworded the description of interrupt facilities (see *Section 21* on page 261). • Reworded the description of interrupt classifications (see *Section 21.1* on page 261). • Reworded the description of interrupt presentation (see *Section 21.2* on page 262). • Modified the term “PPE” with the adjective “logical” in the description of the Internal Interrupt Controller Registers (see *Section 21.3* on page 263). • Revised the description of the Interrupt Pending Port Registers and of the Class and ISRC fields in those registers (see *Section 21.3.1* on page 263). • Corrected the bit range for INT_Generation[Priority] (see *Section 21.3.2* on page 267). • Changed the description of the SPU Halt Instruction Trap or Single Instruction Step Complete interrupt type (see *Table 21-3* on page 269). • Changed the description of class 1 interrupts (see *Section 21.5.2* on page 272) and class 2 interrupts (see *Section 21.5.3* on page 274). • Reworded an implementation note in the description of MFC Interrupt Status Registers (see *Section 21.7* on page 279). • Corrected the meanings of the INT_Stat_class1[LP, LG] values (see *Section 21.7.2* on page 281). • Changed the title of *Table A-1* on page 294. Corrected various other minor errors in that table. • Corrected the description of the SPU_Sig_Notify_2 register (see *Table A-2* on page 296). • Changed a “less than” sign to a “less than or equal to” sign (see *Table A-6* on page 303). • Reworded the description of the **mfceieio** command (see *Table D-4* on page 314). • Added *Table D-5 MFC Atomic Commands* on page 314. • Completely revised *Appendix E Extensions to the PowerPC Architecture* on page 315. Added *Appendix E.1 Software Management of TLBs (optional)* on page 315, *Appendix E.2 Mediated External Exception Extension (optional)* on page 316, *Appendix E.3 Multiple Concurrent Large Pages (optional)* on page 318, *Appendix E.4 Defined Behavior for Inaccessible SPRs* on page 319, and *Appendix E.5 Vector/SIMD Multimedia Extension (optional)* on page 319. Indicated which extensions are required and which are optional. • Updated the Glossary (see page 329). • Corrected various problems in the index (page 341). • Changed SPU_OutIntrMbox to SPU_Out_Intr_Mbox throughout.
August 8, 2005	1.0	Initial release

1. Introduction to Cell Broadband Engine Architecture

The Cell Broadband Engine Architecture (CBEA) defines a processor structure directed toward distributed processing. The intent is to allow implementation of a wide range of single or multiple processor and memory configurations, in order to optimally address many different systems and application requirements.

The CBEA is divided into two environments: the user mode environment (UME) and the privileged mode environment (PME). The UME describes the commands and facilities available to an application programmer; the PME should only be used by software in privileged mode, such as operating systems.

This document does not cover all aspects of the CBEA. The *PowerPC Architecture* documentation defines instructions and facilities for the PowerPC® Processor Element (PPE). The *Synergistic Processor Unit Instruction Set Architecture* document defines instructions and facilities for the synergistic processor unit (SPU). This document focuses on the infrastructure around the computational, data movement, communication, synchronization, and resource management components. Together, these three documents are required for a complete definition of the CBEA.

The PowerPC Architecture™ is a standalone processor architecture. Therefore, this document outlines the optional PowerPC features and extensions required for CBEA compliance. The *Synergistic Processor Unit Instruction Set Architecture* focuses on the instruction set for the SPU component. The SPU is detailed in a separate document so that its instruction set can be used in non-CBEA-compliant implementations and in other architectures.

1.1 Organization of a CBEA-Compliant Processor

Physically, a CBEA-compliant processor can consist of a single chip, a multichip module (or modules), or multiple single-chip modules on a system board or other second-level package. The design depends on the technology used, and on the cost and performance characteristics of the intended design point.

Logically, the CBEA defines four separate types of functional components: the PowerPC Processor Element (PPE), the synergistic processor unit (SPU), the memory flow controller (MFC), and the internal interrupt controller (IIC). The computational units in the CBEA-compliant processor are the PPEs and the SPUs. Each SPU must have a dedicated local storage, a dedicated MFC with its associated memory management unit (MMU), and a replacement management table (RMT). The combination of these components is called a Synergistic Processor Element (SPE).

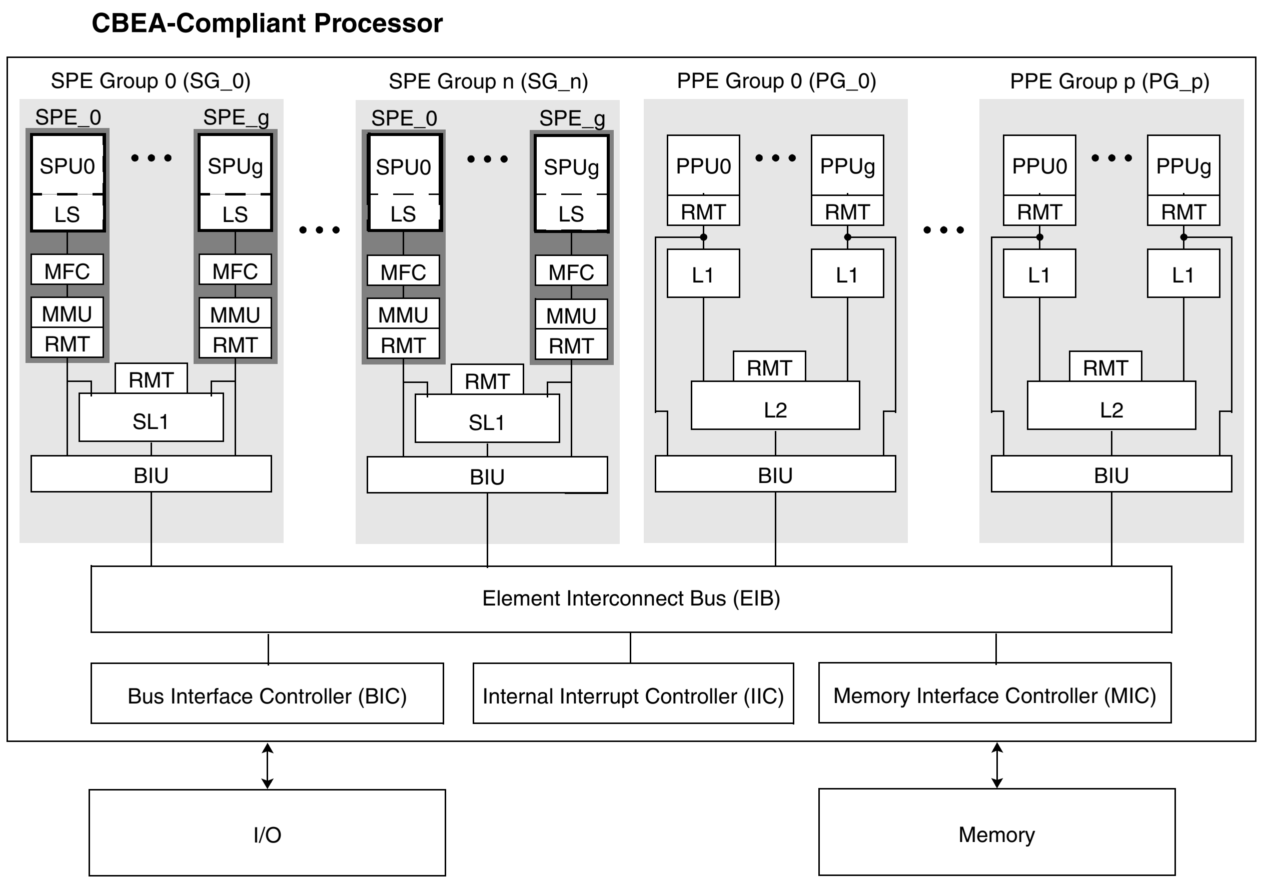
Figure 1-1 on page 28 illustrates a CBEA-compliant processor in which a group of SPEs shares a single SL1 cache. (An SL1 cache is a first-level cache for direct memory access (DMA) transfers between local storage and main storage.) A group of PPEs shares a single second-level (L2) cache. (While *Figure 1-1* shows caches for an SPE and a PPE, they are considered optional in the CBEA.) The illustration also shows two controllers that are typically found in a processor: a memory interface controller (MIC) and a bus interface controller (BIC). An element interconnect bus (EIB) connects the various units within the processor. The requirements for the MIC, BIC, and EIB vary widely between implementations. Thus, the definition for these units is beyond the scope of the CBEA.

A processor can include multiple groups of PPEs (PPE groups) and multiple groups of SPEs (SPE groups). Hardware resources can be shared between units within a group. However, the SPEs and PPEs must appear to software as independent elements.

In summary, a CBEA-compliant system must include the components listed below. Each of these components must follow the definitions of the instructions and facilities provided in the this document, in *PowerPC Architecture, Books I - III,* and in the *Synergistic Processor Unit Instruction Set Architecture* document.

- One or more PPEs
- One or more SPEs, which are the combination of an SPU, a local storage area, an MFC, and an RMT
- One IIC
- One EIB for connecting units within the processor

Figure 1-1. CBEA-Compliant Processor System



BIC	Bus Interface Controller	MMU	Memory Management Unit
BIU	Bus Interface Unit	PPE	PowerPC Processor Element
IIC	Internal Interrupt Controller	PPU	PowerPC processor unit
L1	Memory Cache Internal to the CPU	RMT	Replacement-Management Table
L2	Memory Cache External to the CPU	SL1	First-Level Cache
LS	Local Storage	SPE	Synergistic Processor Element
MFC	Memory Flow Controller	SPU	Synergistic Processor Unit
MIC	Memory Interface Controller		

1.1.1 PowerPC Processor Element

A CBEA-compliant processor includes one or more PPEs. The PPEs are 64-bit PowerPC processor units (PPUs) with associated caches that conform to *PowerPC Architecture, Books I - III.* For more detail on compatibility issues, see *Section 4 PowerPC Processor Element* beginning on page 47. A CBEA-compliant system can include a vector/SIMD multimedia extension unit in a PPE (see *Appendix E.5 Vector/SIMD Multimedia Extension (optional)* on page 319).

The PPEs are general-purpose processing units that can access system management resources (such as the memory-protection tables, for example). Hardware resources defined in the CBEA are mapped explicitly to the real address space seen by the PPEs. Therefore, any PPE can address any of these resources directly by using an appropriate effective address value. A primary function of the PPEs is the management and allocation of tasks for the SPEs in a system.

1.1.2 Synergistic Processor Unit

A CBEA-compliant processor includes one or more SPUs. The SPUs are less complex computational units than PPEs because they do not perform any system management functions. They have a single instruction, multiple data (SIMD) capability. They typically process data and initiate any required data transfers in order to perform their allocated tasks.

The purpose of the SPU is to enable applications that require a higher computational unit density and can effectively use the provided instruction set. Including one or more SPUs in a processor, managed by the PPEs, allows for cost-effective processing over a wide range of applications.

The SPUs implement their own instruction set architecture. The main characteristics of this architecture are described in *Section 5 Synergistic Processor Unit* beginning on page 49.

1.1.3 Memory Flow Controller

MFCs are essentially the data transfer engines. They provide the primary method for data transfer, protection, and synchronization between main storage and the associated local storage, or between the associated local storage and another local storage (see *Section 1.2 Storage Types* on page 30 for an explanation of storage types). An MFC command describes the transfer to be performed. A principal architectural objective of the MFC is to perform these data transfer operations in as fast and as fair a manner as possible, thereby maximizing the overall throughput of a CBEA-compliant processor.

Commands that transfer data are called MFC DMA commands. These commands are converted into DMA transfers between the local storage domain and main storage domain. Each MFC can typically support multiple DMA transfers at the same time and can maintain and process multiple MFC commands.

To accomplish this, the MFC maintains and processes queues of MFC commands. Each MFC provides one queue for the associated SPU (MFC SPU command queue) and one queue for other processors and devices (MFC proxy command queue). Logically, a set of MFC queues is always associated with each SPU in a CBEA-compliant processor. However, some implementations of the architecture can share a single physical MFC between multiple SPUs, such as an SPU group. In such cases, all the MFC facilities must appear to software as independent for each SPU.

Each MFC DMA data transfer command request involves both a local storage address (LSA) and an effective address. The local storage address can directly address only the local storage area of its associated SPU.

The effective address has a more general application. It can refer to main storage, including all the SPU local storage areas, if they are aliased into the real address space (that is, if MFC_SR1[D] is set to ‘1’). (See *Section 15.1 MFC State Register One* beginning on page 221 for more information.)

The MMU in an MFC supports the storage addressing model described in *PowerPC Architecture, Book III* and the extensions described in *Section 14 Storage Addressing* beginning on page 199.

An MFC provides two types of interfaces: one to the SPUs and another to all other processors and devices in a processing group.

- **SPU Channel:** The SPUs use a channel interface to control the MFC. Code running on an SPU can only access the MFC SPU command queue for that SPU.
- **Memory-Mapped Register:** Other processors and devices control the MFC by using memory-mapped registers. It is possible for any processor or device in the system to control an MFC and to issue MFC proxy command requests on behalf of the SPU.

The MFC also supports bandwidth reservation and data synchronization features.

1.1.4 Internal Interrupt Controller

The IIC manages the priority of the interrupts presented to the PPEs. The main purpose of the IIC is to allow interrupts from other components in the processor to be handled without using the main system interrupt controller. The IIC is really a second level controller. It handles all interrupts internal to a CBEA-compliant processor or within a multiprocessor system of CBEA-compliant processors. The system interrupt controller typically handles all interrupts external to the CBEA-compliant processor.

In a CBEA-compliant system, software must first check the IIC to determine if the interrupt was sourced from an external system interrupt controller. The IIC is not intended to replace the main system interrupt controller, which handles interrupts from all I/O devices.

1.2 Storage Types

There are two types of storage domains within the CBEA: the local storage domain and the main storage domain. The local storage of the Synergistic Processor Elements (SPEs) exists in the local storage domain. All other facilities and memory are in the main storage domain.

Local storage consists of one or more separate areas of memory storage, each one associated with a specific SPU. Each SPU can only execute instructions (including data load and data store operations) from within its own associated local storage domain. Therefore, any required data transfers to, or from, storage elsewhere in a system must always be performed by issuing an MFC DMA command to transfer data between the local storage domain of the individual SPU and the main storage domain, unless local storage aliasing is enabled.

1.2.1 Local Storage Addressing

An SPU program references its local storage domain by using a local storage address. Each local storage area is also assigned a real address within the main storage domain. This is called being aliased into the main storage domain. By setting MFC_SR1[D] to ‘1’, privileged software enables an SPE to decode the address space. This lets the local storage be accessed from the main storage domain. With the local storage alias enabled, privileged software can map a local storage area into the effective address space of an application. This allows DMA transfers between the local storage of one SPU and the local storage of another

SPU. Similarly, other processors or devices with access to the main storage domain can also directly access a local storage area that has been aliased into the main storage domain. With the local storage alias enabled, privileged software can map an effective address or I/O bus address to the real address in the main storage domain corresponding to a local storage area.

Data transfers that use the local storage area aliased in the main storage domain should do so as caching inhibited, because these accesses are not coherent with the SPU local storage accesses[1] in the SPU local storage domain. Aliasing the local storage areas into the real address space of the main storage domain allows any other processors or devices that have access to the main storage area direct access to local storage. However, because aliased local storage must be treated as noncacheable, transferring a large amount of data using PPE load and store instructions can result in poor performance. Data transfers between the local storage domain and the main storage domain should use the MFC DMA commands to avoid stalls.

1.2.2 Main Storage Addressing

The addressing of main storage in the CBEA is compatible with the addressing defined in the PowerPC Architecture. The CBEA builds upon the concepts of the PowerPC Architecture and extends them to the addressing of main storage by the MFCs.

An application program executing in an SPU or in any other processor or device uses an effective address to access main storage. The effective address is computed when a PPE performs a load, store, branch, or cache instruction, and when it fetches the next sequential instruction. An SPU program must provide the effective address as a parameter in an MFC command. The effective address is translated to a real address according to the procedures described in the overview of address translation in *PowerPC Architecture, Book III*. The real address is the location in main storage that is referenced by the translated effective address.

All PPEs, MFCs, and I/O devices in a system share main storage. All information held in this level of storage is visible to all processors and devices in the system. This storage area can either be uniform in structure or can be part of a hierarchical cache structure. Programs reference this level of storage by using an effective address.

1.2.3 Main Storage Attributes

The main storage of a system typically includes both general-purpose and nonvolatile storage. It also includes special-purpose hardware registers or arrays used for functions such as system configuration, data-transfer synchronization, memory-mapped I/O, and I/O subsystems.

Table 1-1 lists the sizes of address spaces in main storage.

Table 1-1. Sizes of Main Storage Address Spaces (Page 1 of 2)

Address Space	Size	Description
Real Address Space	2^m bytes	where $m \leq 62$
Effective Address Space	2^{64} bytes	An effective address is translated to a virtual address using the segment lookaside buffer (SLB).
Virtual Address Space	2^n bytes	where $65 \leq n \leq 80$ A virtual address is translated to a real address using the page table.
Note: The values of "m," "n," and "p" are implementation dependent.		

1. SPU load, store, or instruction fetch

Table 1-1. Sizes of Main Storage Address Spaces (Page 2 of 2)

Address Space	Size	Description
Real Page (Base)	2^{12} bytes	
Virtual Page	2^p bytes	where $12 \le p \le 28$ Up to eight page sizes can be supported simultaneously. A small 4 KB (p = 12) page is always supported. The number of large pages and their sizes are implementation dependent.
Segment	2^{28} bytes	The number of virtual segments is $2^{(n - 28)}$ where $65 \le n \le 80$
Note: The values of “m,” “n,” and “p” are implementation dependent.		

1.3 Cache Replacement Management Facility

The CBEA includes an optional facility for managing critical resources within the processor and system. The resources targeted for management within the CBEA are the translation lookaside buffers (TLBs) and the data and instruction caches. Implementation-dependent tables control management of these resources. Tables for managing TLBs and caches are called replacement management tables (RMTs). These tables are optional in the CBEA. However, it is strongly recommended that an implementation provide a table for each critical resource that can be a bottleneck in the system.

An SPE group can also contain an optional cache hierarchy, the SL1 caches, which represent first level caches for DMA transfers. The SL1 caches can also contain optional RMTs. *Section 7.7 Storage Control Commands* beginning on page 67 describes the features and operation of the SL1 caches. *Section 19 Cache Replacement Management Facility* beginning on page 255 describes the RMTs.

1.4 Instructions, Commands, and Facilities

Instructions define operations to be performed by a processing element. The PowerPC Architecture describes the instructions supported for a PPE. The CBEA requires certain features and extensions that are optional in the PowerPC Architecture. *Section 4 PowerPC Processor Element* beginning on page 47 and *Section 13 PowerPC Architecture, Book III Compatibility* beginning on page 197 describe these features and extensions. *Section 5 Synergistic Processor Unit* beginning on page 49 defines the version of the Synergistic Processor Instruction Set Architecture supported by the CBEA.

Commands define operations to be performed by the MFC. *Section 7 MFC Commands* beginning on page 55 describes the supported commands.

The term *facilities* describes functionality accessed through the main storage domain by processors or devices having such access, or by SPUs through the use of channel instructions. In some cases, a facility can be accessed through a single main storage address or by a single SPU channel. *Section 6.1 MFC Facilities* beginning on page 52 lists the facilities supported by the CBEA.

1.5 Reserved Fields and Registers

All reserved fields should be set to zero. MFC commands with reserved fields that are not set to zero are considered invalid. For a description of the invalid instructions and invalid MFC forms, see *Section 7 MFC Commands* beginning on page 55.

The handling of reserved bits in a specific instantiation of the CBEA is implementation dependent. For each reserved bit, an implementation will either:

- Ignore the reserved bits on writes and return zeros for the reserved bits on reads; or
- Maintain the state of the reserved bits.

All reserved registers should be set to zero, unless otherwise indicated. An implementation must decode all reserved registers for both reads and writes. The handling of reserved register values in a specific instantiation of the CBEA is implementation dependent. For each reserved register, an implementation will take one of the following actions:

- Ignore the value on writes and return zeros for reads of the register
- Maintain the state of the reserved register

Fields and registers that are currently defined as reserved can become defined in future versions of the CBEA.

Programming Note:

Software must preserve the state of the reserved bits. To accomplish this in an implementation-independent fashion, software should take the following steps:

- Initialize all reserved bits to zero.
- Alter only the defined bits by reading the register, modifying the required bits, and writing the new value back to the register.

See the register definition for the proper method of handling reserved bits. Software should not use reserved bits or reserved registers to maintain the software state for any purpose.

Implementation Note:

An implementation of the CBEA should never use reserved fields or registers for an implementation-dependent purpose. All defined, reserved, and implementation-dependent registers must be decoded for both reads and writes. Furthermore, the range of addresses for a given area in the memory map must be a multiple of at least 4 KB (that is, the smallest page addresses). An implementation should consider the range to be a multiple of a larger supported page size to minimize the number of page table entries required to map the address range. See *Appendix A Memory Maps* on page 293 for more details.

1.6 Implementation-Dependent Fields and Registers

The CBEA provides implementation-dependent fields and registers. These fields and registers are intended for implementation-dependent purposes and will not be defined by future versions of the CBEA. Unused fields and registers should be handled in the same manner as reserved fields and registers.

For descriptions of the implementation-dependent fields and registers, see the specific implementation documentation.

User Mode Environment

The *User Mode Environment (UME)* section defines the instruction set, base command set, storage models, and facilities available to an application programmer. It also discusses compatibility with the PowerPC Architecture. In addition to an overview, this section includes:

- *Section 3 Storage Models* beginning on page 41
- *Section 4 PowerPC Processor Element* beginning on page 47
- *Section 5 Synergistic Processor Unit* beginning on page 49
- *Section 6 Memory Flow Controller* beginning on page 51
- *Section 7 MFC Commands* beginning on page 55
- *Section 8 Problem-State Memory-Mapped Registers* beginning on page 79
- *Section 9 Synergistic Processor Unit Channels* beginning on page 113
- *Section 10 Storage Access Ordering* beginning on page 175
- *Section 11 SPU Isolation Facility* beginning on page 183

2. Overview

Introduction to Cell Broadband Engine Architecture on page 27 reviews the structure of a Cell Broadband Engine Architecture (CBEA)-compliant system. Become familiar that section and *Figure 1-1 CBEA-Compliant Processor System* on page 28 to gain an understanding of the following topics:

- PowerPC Processor Element (PPE)
- Synergistic processor unit (SPU)
- Memory flow controller (MFC)
- System local storage and main storage memory arrays, their associated cache structures, and their organization within an overall system context

The instructions and facilities provided by a PPE are defined in *PowerPC Architecture, Books I-III*. The instructions and facilities provided by an SPU are defined in the *Synergistic Processor Unit Instruction Set Architecture* document. For a more complete understanding of the CBEA, the reader should also become familiar with these documents.

2.1 Instruction and Command Classes

Both the PPE and the SPU components execute programs that consist of instructions that specify the type of actions they are to perform. The MFCs execute commands that specify the type of data copying or movement they are to perform. Thus, there are generally two sets of instructions (PPE and SPU) and one set of commands (MFC). These instructions and commands can be categorized into three classes.

- Defined
- Illegal
- Reserved

The class of an instruction or command is determined by examining the operation code (opcode) and, if it exists, the extended opcode. If an instruction opcode, or a combination of opcode and extended opcode, is not that of a defined or reserved instruction, then the instruction is illegal. If the command type is not that of a defined or reserved command, then the command is illegal.

A given instruction or command is in the same class for all implementations compliant with this release of the CBEA. In future versions of the CBEA, instructions or commands that are currently illegal can become defined (by being added to the architecture) or reserved (by being assigned to a special-purpose operation). Similarly, some instructions or commands that are currently reserved can become defined in a subsequent architecture release.

2.1.1 Defined Class

This class of instructions and commands contains all instructions and commands defined in this release of the CBEA. In general, defined instructions and commands are guaranteed to be provided in all implementations. The only deviations permitted are instructions or commands that are specifically identified in their descriptions as optional. Defined instructions or commands can have preferred forms, or invalid forms, or optional forms. *Section 2.2 Forms of Defined Instructions and Commands* on page 37 describes these forms.

2.1.2 Illegal Class

Illegal instructions and commands are available for use in future extensions of the CBEA. This means that a future release of the CBEA might assign any of these opcodes to new instructions or functions. *PowerPC Architecture, Book I* describes illegal PPE instructions. *Section 7.1.2 Illegal Commands* beginning on page 60 describes illegal MFC commands.

Any attempt to execute an illegal PPE instruction causes an exception interrupt but has no other effect on PPE operation. Any SPU that encounters an error when executing an instruction immediately halts program execution, records the event in its status register, and requests an external interrupt. The illegal-instruction interrupt should be enabled and routed to a PPE. In either case, the exception interrupt should cause the illegal-instruction handler for the system to be invoked; the illegal-instruction handler then takes appropriate action.

Any PPE instruction that consists entirely of binary zeros is an illegal instruction. In an SPU instruction, an opcode of zero is a stop instruction, which causes SPU execution to stop. This increases the probability that any attempt to execute data or uninitialized storage invokes the system illegal-instruction interrupt handler.

The MFC commands are not fully checked when enqueued. Therefore, an illegal MFC command might only be recognized as it is taken off the queue for execution, asynchronously to when it was enqueued. Consequently, illegal MFC commands are included with, and treated in the same manner as, other causes of MFC or direct memory access (DMA) asynchronous exceptions (see *Section 7.2* beginning on page 61).

In general, all exceptions, including illegal MFC commands and other DMA processing errors, cause the associated command queue processing to suspend as soon as they are detected. The exceptions also cause an exception interrupt to be generated and sent to a PPE. *Section 2.3 Exceptions* beginning on page 38 gives an overview of the various ways in which these exception interrupts can be generated.

2.1.3 Reserved Class

Reserved instructions are allocated to specific purposes outside the scope of the CBEA or are intended for use in future extensions of the CBEA. *PowerPC Architecture, Book I* defines the reserved PPE instructions. *Section 7 MFC Commands* beginning on page 55 defines the reserved MFC commands. These are the only commands that implementation-dependent applications should use. The specific implementation documentation describes how attempts to execute reserved instructions and commands are handled. If the use of a reserved instruction or command is not covered in the specific implementation documentation, it is treated as an illegal command.

2.2 Forms of Defined Instructions and Commands

In the defined set of instructions and commands, certain field or parameter settings can execute more efficiently, or can produce an error condition. The CBEA defines the field and parameter settings as preferred forms or invalid forms.

2.2.1 Preferred Forms

There are no preferred forms of the SPU instructions or of the MFC commands. The preferred forms of the PPE instructions are defined in the *PowerPC Architecture*.

2.2.2 Invalid Forms

Some defined instructions and commands have invalid forms. An instruction or command is considered invalid if one or more fields (excluding those that specify the operation) are coded in a manner that can be deduced as incorrect by examining that encoding. *PowerPC User Instruction Set Architecture, Book I* defines invalid PPE instruction forms. *Section 7.1 Command Classes* beginning on page 57 defines invalid DMA MFC commands.

2.2.3 Optional Forms

Some of the defined instructions are optional. Any attempt to execute an optional instruction that is not provided by the implementation causes the system illegal-instruction interrupt handler to be invoked.

A facility, instruction, or command can be optional for any of the following reasons:

1. It is being phased into the architecture. At some future date, it will be required and no longer optional.
2. It is being phased out of the architecture. System developers should develop a migration plan to eliminate its use in new systems.
3. It is useful primarily for certain kinds of applications and systems. It is likely to remain in the architecture, as optional.

Reasons 1 and 2 permit the architecture to evolve gradually by providing an intermediate status for facilities and instructions that are being added to or removed from the architecture. Reason 3 is intended for facilities and instructions that are typically used primarily in library routines.

Currently, there are no optional commands. The *Synergistic Processor Unit Instruction Set Architecture* document defines five optional, double-precision SPU instructions:

- Double Floating Compare Equal (**dfceq**)
- Double Floating Compare Magnitude Equal (**dfcmeq**)
- Double Floating Compare Greater Than (**dfcgt**)
- Double Floating Compare Magnitude Greater Than (**dfcmgt**)
- Double Floating Test Special Value (**dftsv**)

There is also an optional facility: the isolation facility.

2.2.4 Optional Fields

Optional fields in the MFC commands are assumed to be zero if not explicitly set. Software does not have to set the optional fields if zeros achieve the required results. For a detailed description of the MFC commands, see *Section 7 MFC Commands* beginning on page 55.

2.3 Exceptions

Note: For a more detailed description of exceptions, see the specific implementation documentation.

Exceptions are the result of an operation that cannot be executed as requested. In the CBEA, there are four types of exceptions:

- Exceptions caused directly by the execution of a PPE instruction
- Exceptions caused by the execution of an SPU instruction

- Exceptions caused by the execution of an MFC DMA command
- System-caused, asynchronous, external-event exceptions.

An exception can set status information in a register. It can also cause an interrupt handler of the system software in a PPE to be invoked.

Exceptions caused by the execution of a PPE instruction are defined in *PowerPC Architecture, Book I*. In the *PowerPC Architecture*, there are only two types of exceptions: those caused by the execution of a PPE instruction and those caused by an asynchronous event. In most cases, the invocation of an interrupt handler for exceptions caused by the execution of a PPE instruction is precise. (That is, the exception is created when the event happens. The PPE instruction that caused the exception is known.) This is not true for floating-point exceptions, when the floating-point exceptions mode is set to one of the imprecise modes. The invocation of an interrupt handler for asynchronous events is always imprecise. (See *PowerPC Architecture, Book I* for more information about PPE instruction-related exceptions.)

Exceptions caused by the execution of an SPU instruction are defined in the *Synergistic Processor Unit Instruction Set Architecture* document and *Section 21 Interrupt Facilities* beginning on page 261. Exceptions caused by the execution of an MFC command are defined in *Section 21 Interrupt Facilities* beginning on page 261. These exceptions generate interrupts in the CBEA. They are typically sent to a PPE as an imprecise external interrupt where they invoke a privileged software interrupt handler.

Exceptions generated directly by the execution of an instruction include:

- The exceptions caused by the execution of a PowerPC instruction, as described in the *PowerPC Architecture, Book I*
- An SPU error that occurs when an instruction is executed
- The execution of a write channel (**wrch**) instruction to the SPU Write Outbound Interrupt Mailbox Channel by the SPU
- The execution of an SPU stop-and-signal instruction

The exceptions generated by an MFC command include:

- An attempt to execute an illegal MFC command
- An attempt to execute a defined MFC command using an invalid form (that is, invalid parameters)
- An attempt to execute a defined MFC command with an alignment error
- The execution of an optional MFC command not supported by the implementation
- An attempt to access storage not defined by the MFC translation facility

2.4 SPU Events

The SPU supports an event facility that enables waiting for or polling for specific events. The event facility can also generate an SPU asynchronous interrupt for specific events. If SPU interrupts are enabled, an occurrence of an unmasked event results in an SPU interrupt handler being invoked with the first instruction of the interrupt handler located at local storage address ‘0’.

For more details, see *Section 9.11 SPU Event Facility* on page 150. Also see the *Synergistic Processor Unit Instruction Set Architecture* for information about enabling, disabling, and handling program code executing on an SPU.

3. Storage Models

A CBEA-compliant processor implements two concurrent storage models for an application program: the virtual storage model of a PPE (also used by MFCs for DMA operations) and the local storage model of an SPU. The PPE virtual storage model allows privileged software to provide the same or different views of the real memory and I/O devices for the PPEs and SPEs in a processor. It is possible for multiple virtual address spaces to exist. The SPU local storage model is restricted to applications running on SPUs and data transfers handled by the MFC.

3.1 Virtual Storage Model

The virtual storage model implements a virtual storage model for applications. This means that a combination of hardware and software can present a storage model that allows applications to exist within a virtual address space larger than either the effective address space or the real address space.

Each program can access 2^{64} bytes of effective address space, subject to limitations imposed by privileged software. In a typical CBEA-compliant processor system, the effective address space of each program is a subset of a larger virtual address space managed by privileged software.

Each effective address is translated to a real address (an address of a byte in real storage or a byte on an I/O device) before being used to access storage. The hardware uses the memory management unit (MMU) address translation facility to accomplish this (for more information, see *PowerPC Architecture, Book III*). The privileged software manages the real storage resources of the system by setting up the tables and other information used by the hardware address translation facility.

The user mode environment deals primarily with effective addresses that are translated by the MMU address translation facility. Each effective address lies in a virtual page.[1] The virtual page is mapped to a real page (4 KB virtual page) or to a contiguous sequence of real pages (large virtual page) before data or instructions in the virtual page are accessed.

In general, real storage might not be large enough to map all the virtual pages used by active applications. With hardware support, the privileged software can attempt to use the available real pages to map a set of virtual pages that is sufficient for the applications. If a sufficient set of virtual pages is maintained, “paging” activity is minimized. If sufficient virtual pages are not available, performance degradation is likely.

Based on system standards and application requests, the privileged software can restrict access to virtual pages. Access to the virtual pages can be read/write, read only, or no access. For example, program code might be designated read only.

See *PowerPC Architecture, Book III* for a complete description of the virtual storage model.

3.2 SPU Local Storage Model

Each SPU has its own dedicated area of local storage. Applications running on a given SPU can only reference the associated local storage area by using a local address for instruction fetch, data load, and data store operations. The individual local storage areas can be aliased to a real address within the main storage domain. Any PPE can access these areas by using the appropriate effective address.

1. Page: An aligned unit of storage for which protection and control attributes can be specified independently, and for which reference and change status are independently recorded.

MFC units process data transfers, which move data between a local address and an effective address. The local address always references the local storage area associated with the MFC. However, the effective address can be arranged to reference any area in the main storage domain, including aliased local storage areas, if required.

3.2.1 Local Storage Access

The CBEA allows the local storage of an SPU to have an alias in the real address space in the main storage domain. This allows other processors in the main storage domain to access local storage through appropriately mapped effective address space. It also allows external devices, such as a graphics device, to directly access the local storage.

3.2.1.1 Mapping Requirements

Privileged software should access the aliased pages of local storage in the main storage domain. If not accessed as caching inhibited, software must explicitly manage the coherency of local storage with other system caches.

3.2.1.2 Local Storage Access Exceptions

MFC commands that access an effective address range that maps to its own local storage can produce an error or unpredictable results. This occurs when the translated effective address area for a DMA overlaps the local storage range of a DMA. If the two address ranges (translated effective address and local storage address) overlap and the source is a lower address than the destination, the DMA results in the corruption of the source data. Address overlap is not detectable and does not generate an exception. Therefore, it is the programmer's and privileged software's responsibility to avoid an unintended overlap.

3.3 Single-Copy Atomicity

An access is single-copy atomic, or just atomic, if it is always performed in its entirety with no visible fragmentation. Atomic accesses are thus serialized. Each happens in its entirety in some order, even when that order is not specified in the program or enforced between processors. In the PowerPC Architecture, the following single register accesses are always atomic:

- All byte accesses
- Halfword accesses aligned on halfword boundaries
- Word accesses aligned on word boundaries
- Doubleword accesses aligned on doubleword boundaries
- Quadword accesses aligned on quadword boundaries
- Cache line accesses by DMA commands aligned on cache line boundaries

No other accesses are guaranteed to be atomic.

An access that is not atomic is performed as a set of smaller, disjointed atomic accesses. The number and alignment of these accesses are implementation dependent, as is the relative order in which they are performed.

In the CBEA, DMA accesses in the main storage domain are atomic if they meet the requirements of the PowerPC Architecture. All other DMA transfers, if greater than a quadword or unaligned, are performed as a set of smaller, disjointed atomic accesses. The number and alignment of these accesses are implementation dependent, as is the relative order in which they are performed. Only quadword accesses of local storage are atomic.

3.4 Cache Models

A cache model in which there is one cache for instructions and another cache for data is called a “Harvard-style” cache. This is the model assumed by the PowerPC Architecture. Alternative cache models can be implemented (such as, a “combined cache” model, in which a single cache is used for both instructions and data, or a model in which there are several levels of caches). However, they must support the programming model implied by a Harvard-style cache.

The processor is not required to maintain copies of storage locations in the instruction cache consistent with modifications to those storage locations (such as, modifications caused by store instructions). A location in the data cache is considered to be modified in that cache if the location has been modified (for example, by a store instruction) and the modified data has not been written to main storage.

Cache management instructions allow programs to manage the caches when needed. For example, program management of the caches is needed when a program generates or modifies code that is executed (that is, when the program modifies data in storage and then attempts to execute the modified data as instructions). The cache management instructions are also useful in optimizing the use of memory 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.

The cache management instructions allow programs to perform the following functions:

- Invalidate the copy of storage in an instruction cache block (**icbi**)
- Provide a hint that the program will probably soon access a specified data cache block (**dcbt**, **dcbtst**)
- Set the contents of a data cache block to zeros (**dcbz**)
- Copy the contents of a modified data cache block to main storage (**dcbst**)
- Copy the contents of a modified data cache block to main storage and make the copy of the block in the data cache invalid (**dcbf**)

The data cache commands in the first level DMA transfer cache (SL1) allow programs to perform the following functions:

- Bring a range of effective addresses into the SL1 (**sdcrt** and **sdcrtst**)
- Write zeros to the contents of a range of effective addresses (**sdcrz**)
- Store the modified contents of a range of effective addresses (**sdcrst**)
- Store the modified contents of a range of effective addresses and invalidate the block (**sdcrf**)

3.5 Memory Coherence

Memory coherence refers to the ordering of stores to a single location. Atomic stores to a given location are coherent if they are serialized in some order, and no processor or no device can observe any subset of those stores as occurring in a conflicting order. This serialization order is an abstract sequence of values; the real storage location need not assume each of the values written to it. For example, a processor can update a location several times before the value is written to real storage.

The result of a store operation is not available to every processor or to every device at the same instant. A processor or device can observe only some of the values that are written to a location. However, when a location is accessed atomically and coherently by all processors and devices, the sequence of values loaded from the location by any processor or any device during any interval of time forms a subsequence of the sequence of values that the location logically held during that interval. That is, a processor or device can never load a newer value first and load an older value later.

Memory coherence is managed in blocks called coherence blocks. Their size is implementation dependent but is typically larger than a word and often the size of a cache block.

For storage that does not require memory coherence, software must explicitly manage memory coherence to the extent required by program correctness. The operations required to do this can be system dependent.

Programming Note:

In most systems, the default is that all storage is memory coherence required. For some applications in some systems, software management of coherence can yield better performance. In such cases, a program can request that a given unit of storage not be memory coherence required. It can manage the coherence of that storage by using the **sync** instruction, the cache management instructions, and services provided by the operating system.

3.6 Storage Control Attributes

Storage control attributes are associated with units of storage that are multiples of the page size. Each storage access is performed according to the storage control attributes of the specified storage location. The storage control attributes are:

- Write through required
- Caching inhibited
- Memory coherence required
- Guarded

These attributes have meaning only when an effective address is translated by the processor performing the storage access. All combinations of these attributes are supported except write through required with caching inhibited.

Programming Note:

The write through required and caching inhibited attributes are mutually exclusive because the write through required attribute permits the storage location to be in the data cache while the caching inhibited attribute does not. Storage that is write through required or caching inhibited is not intended to be used for general-purpose programming. For example, the **lwarx**, **ldarx**, **stwcx.**, and **stdcx.** PowerPC instructions and the **sdcrt**, **sdcrtst**, **sdcrz**, **sdcrst**, and **sdcrf** MFC commands can cause the interrupt handler for system data storage to be invoked if they specify a location in storage that has either of these attributes.

3.7 Shared Storage

The CBEA supports the sharing of storage between programs, between different instances of the same program, between SPUs, and between processors and other devices. It also supports access to a storage location by one or more programs using different effective addresses or DMA addresses. All these cases are considered storage sharing. Storage is shared in blocks of an integral number of pages.

When the same storage location has different effective addresses, the addresses are called aliases. Each application can be granted separate access privileges to aliased pages.

4. PowerPC Processor Element

The Cell Broadband Engine Architecture (CBEA) includes PowerPC processors known as PowerPC Processor Elements (PPEs). *PowerPC Architecture, Books I- III* defines the architecture of a PPE.

A PPE must be a 64-bit implementation, in which all effective addresses and registers, except some special-purpose and memory-mapped I/O (MMIO) registers, are 64 bits long. All implementations have two modes of operation: 64-bit mode and 32-bit mode. The mode controls how the effective address is interpreted, how status bits are set, and how the count register is tested by branch-conditional instructions. All instructions are available in both modes. In both 64-bit mode and 32-bit mode, instructions that set a 64-bit register affect all 64 bits. The value placed in the register is independent of mode. In both modes, effective address computations use all 64 bits of the relevant registers (such as, the general-purpose registers, link register, and count register) and produce a 64-bit result. However, in 32-bit mode, the high-order 32 bits of the computed effective address are ignored when accessing data and are set to zero when fetching instructions.

The CBEA does not permit a PPE implementation that provides only the equivalent of 32-bit mode (an implementation in which all registers except floating-point registers are 32 bits long).

4.1 PowerPC Architecture, Book I and Book II Compatibility

The CBEA user mode environment is compatible with *PowerPC Architecture, Book I.* The CBEA privileged mode environment is compatible with *PowerPC Architecture, Book II.* A PPE provides binary compatibility for PowerPC applications, except as described in *Section 4.1.2 Incompatibilities with PowerPC Architecture, Book I*.

The CBEA does *not* automatically track changes and extensions to the PowerPC Architecture. Inclusion of changes and extensions to the PowerPC Architecture will be identified in future version of the CBEA.

4.1.1 Optional Features in PowerPC Architecture, Book I (Required for CBEA)

The following instructions are considered optional in the PowerPC Architecture but are required for a PPE by the CBEA user mode environment:

- Floating reciprocal estimate single A-form (**fres**)
- Floating reciprocal square-root estimate A-form (**frsqte**)

The **fres** and **frsqte** instructions are optional floating-point instructions in the PowerPC Architecture. These instructions are a subset of the graphics group. An implementation cannot claim support for the graphics group without implementing the remaining instructions in this group. *PowerPC Architecture, Book I* describes these instructions.

Note: The optional PowerPC floating-point instructions that are mandatory in the CBEA are needed for the application space targeted by the CBEA.

4.1.2 Incompatibilities with PowerPC Architecture, Book I

There are no incompatibilities with *PowerPC Architecture, Book I*.

4.1.3 Optional Features in PowerPC Architecture, Book II (Required for CBEA)

The following instruction is considered optional in the PowerPC Architecture but is required in the CBEA:

- Data cache block touch X-form (**dcbt**)

  This optional version of the data cache block touch instruction permits a program to provide a hint regarding a sequence of contiguous data cache blocks. Such a sequence is called a “data stream.” A **dcbt** instruction in which TH[0:3] is not equal to x'0' is said to be a “data stream variant” of **dcbt**. For more information on the optional version of the **dcbt** instruction, see Book II of the *PowerPC Architecture*. For more details on the variants of the **dcbt** instruction supported by an implementation, see the specific implementation documentation.

4.1.4 Incompatibilities with PowerPC Architecture, Book II

There are no incompatibilities with *PowerPC Architecture, Book II*.

4.1.5 Extensions to the PowerPC Architecture, Books I and II

The following facility is considered optional in *PowerPC Architecture*, *Books I* and *II* but is required for compliance with the Cell Broadband Engine Architecture:

- Vector/SIMD multimedia extension (optional) (see page 319)

5. Synergistic Processor Unit

The intent of the synergistic processor unit (SPU) is to fill a void between general-purpose processors and special-purpose hardware. General-purpose processors aim to achieve the best average performance on a broad set of applications. Special-purpose hardware aims to achieve the best performance on a single application. The SPU, however, aims to achieve leadership performance on critical workloads for game, media, and broadband systems. The intent of the SPU and the Cell Broadband Engine Architecture (CBEA) is to provide a high degree of control to expert (real-time) programmers while maintaining ease of programming.

The SPU implements a its own instruction set architecture (ISA). This architecture is described in a separate document, the *Synergistic Processor Unit Instruction Set Architecture*.

The main characteristics of this architecture are:

- Load-store architecture with sequential semantics, using a set of 128 registers, each of which is 128 bits wide.
- Single-instruction, multiple-data (SIMD) capability
  - Sixteen 8-bit integers
  - Eight 16-bit integers
  - Four 32-bit integer or four single-precision floating-point values
  - Two double-precision floating point
- SPU load and store instructions access only the associated local storage register
- Channel input/output for memory flow controller (MFC) control (used for external data access)

The SPU has the following restrictions:

- No direct access to main storage (access to main storage using MFC facilities only)
- No distinction between user mode and privileged state
- No access to critical system control such as page-table entries (PowerPC Processor Element [PPE] privileged software should enforce this restriction)
- No synchronization facilities for shared local storage access

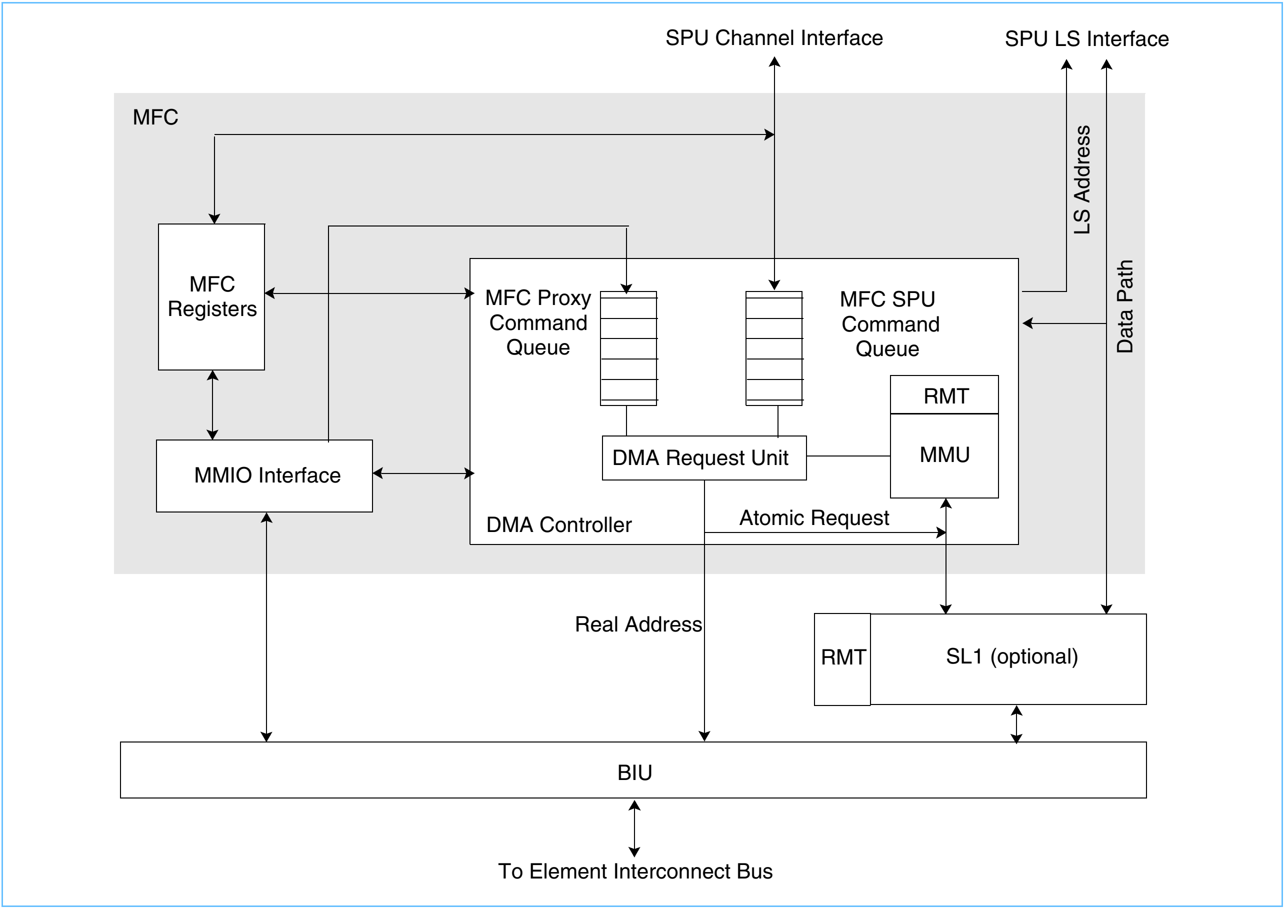
The intent of the SPU is to enable applications that require a high computational unit density and that can effectively use the instruction set provided. A significant number of SPU cores in a system, managed by a PPE, allows for cost-effective processing over a wide range of applications.

6. Memory Flow Controller

In a Cell Broadband Engine Architecture (CBEA)-compliant processor, the memory flow controller (MFC) serves as an interface to the system and to other elements for a synergistic processor unit (SPU). It provides the primary mechanism for data transfer, protection, and synchronization between main storage and the local storage arrays. As discussed in *Section 1.1 Organization of a CBEA-Compliant Processor* on page 27, there is logically an MFC for each SPU in a processor. Some implementations can share resources of a single MFC between multiple SPUs. In this case, all the facilities and commands defined for the MFC must appear independent to software for each SPU. The effects of sharing an MFC must be limited to implementation-dependent facilities and commands.

Figure 6-1 shows a high-level block diagram of a typical MFC. In this illustration, the MFC has two interfaces to the SPU, two interfaces to the bus interface unit (BIU), and two interfaces to an optional first level DMA transfer cache (SL1). The SPU interfaces are the SPU channel interface and the SPU local storage (LS) interface. The SPU channel interface allows the SPU to access MFC facilities and to issue MFC commands. The MFC uses the SPU local storage interfaces to access local storage in the SPU. One interface to the BIU allows memory-mapped I/O (MMIO) access to the MFC facilities. This interface also allows other processors to issue MFC commands. Commands issued using MMIO are called MFC proxy commands. The other interface to the BIU carries the real address. The interfaces to the SL1 cache are mainly for data transfers. The MFC uses one interface for access to the address translation tables in main storage. The other interface to the SL1 cache transfers data between main storage and local storage.

Figure 6-1. Typical MFC Block Diagram



As shown in *Figure 6-1* on page 51, a typical MFC consists the following main units:

- MMIO interface
- MFC registers
- Direct memory access (DMA) controller

The MMIO interface maps the MFC facilities of the SPU into the real address space of the system. This allows access to the MFC facilities from any processor or any device in the system. In addition, the MMIO interface can be configured to map the local storage of the SPU into the real address space. This allows direct access to the local storage from any processor or any device in the system that enables local-storage-to-local-storage transfers and lets I/O devices directly access the local storage domain of an SPU. Coherency is not maintained between SPU and MMIO accesses to the local storage domain.

6.1 MFC Facilities

Most of the MFC facilities are contained in the MFC Registers unit. Some facilities are contained in the direct memory access controller (DMAC). A list of the facilities within the MFC follows.

User mode environment facilities include:

- Proxy Tag-Group Completion Facility (see page 92)
- SPU Control and Status Facilities (see page 96)
- Mailbox Facility (see page 101)
- SPU Signal Notification Facility (see page 105)
- MFC Multisource Synchronization Facility (see page 108)
- SPU Isolation Facility (see page 183)

Privileged mode environment facilities include:

- MFC Privileged Facilities (see page 221)
  - MFC State Register One (see page 221)
  - MFC Logical Partition ID Register (see page 223)
  - MFC Storage Description Register (see page 224)
  - MFC Data Address Register (see page 225)
  - MFC Data Storage Interrupt Status Register (see page 226)
  - MFC Address Compare Control Register (see page 227)
  - MFC Local Storage Address Compare Facility (see page 229)
  - MFC Command Error Register (see page 231)
  - MFC Data Storage Interrupt Pointer Register (see page 232)
  - MFC Control Register (see page 233)
  - MFC Atomic Flush Register (see page 236)
  - SPU Outbound Interrupt Mailbox Register (see page 237)
- SPU Privileged Facilities (see page 239)
  - SPU Privileged Control Register (see page 239)
  - SPU Local Storage Limit Register (see page 241)
  - SPU Configuration Register (see page 245)
- SPE Context Save and Restore (see page 247)

Synchronization and the transfer of data are generally the responsibility of the DMAC within the MFC.
The DMAC can move data between the local storage of an SPU and the main storage area. Optionally, the data can be cached in the SL1.

The Synergistic Processor Elements (SPEs) and the PowerPC Processor Elements (PPEs) instruct the MFC to perform these DMA operations by queuing DMA command requests to the MFC through one of the command queues:

- Commands issued by an SPE are queued to the MFC SPU command queue.
- Commands issued by a PPE are queued to the MFC proxy command queue.

The MFC uses a memory management unit (MMU) to perform all MFC address translations and MFC access protection checks required for the DMA transfers. The MMU handles MFC transfers in much the same way that a PPE handles load and store operations.

Ordering of the data transfers for a given command adheres to the rules specified in the PowerPC Architecture. *Section 7.9 MFC Synchronization Commands* beginning on page 73 describes the ordering between commands.

Section 14 Storage Addressing beginning on page 199 describes the memory management facilities of a CBEA-compliant processor.

Section 7.5 Get Commands (Main Storage to Local Storage) beginning on page 64 and *Section 7.6 Put Commands (Local Storage to Main Storage)* beginning on page 65 describe the DMA data transfer commands available to the application programmer. The MFC employs a 64-bit effective address field, which is translated and then used to reference main storage. A 32-bit address field is used to reference local storage directly.

7. MFC Commands

Memory flow controller (MFC) commands enable code executing in a synergistic processor unit (SPU) to access main storage and maintain synchronization with other processors and devices in the system. Commands are also provided to manage optional caches.

Code running on an SPU can issue MFC commands directly. MFC commands issued by an SPU are called “MFC SPU commands”. Code running on another processor or device, such as a PowerPC Processor Element (PPE), can issue MFC commands on behalf of an SPU. MFC commands issued by a PPE or another processor or device are called “MFC proxy commands.”

MFC SPU commands—Code running on an SPU executes a series of channel instructions to issue an MFC SPU command. MFC SPU commands are queued to the MFC SPU command queue.

MFC proxy commands—Code running on other processors or devices performs a series of memory-mapped I/O (MMIO) transfers to issue an MFC proxy command for an SPE. MFC proxy commands are queued to an MFC proxy command queue.

MFC commands that transfer data are called “MFC DMA commands.” The data transfer direction for MFC direct memory access (DMA) commands is always referenced from the perspective of an SPE. Therefore, commands that transfer data into an SPE (from main storage to local storage) are considered **get** commands. Commands that transfer data out of an SPE (from local storage to main storage) are considered **put** commands.

The following command modifiers are associated with the MFC DMA commands. These command modifiers extend or refine the function of a command. For example, a **put** command moves data from local storage to an effective address within the main storage domain. A **puts** command moves data from local storage to an effective address within the main storage domain and starts the SPU after the DMA operation completes.

s	Starts the execution of the SPU at the current location indicated by the SPU Next Program Counter Register after the data has been transferred into or out of the local storage.
r	Performance hint for DMA put operations. The hint is intended to allow another processor or device, such as a PPE, to capture the data into its cache.
f	Tag-specific fence. A command with a tag-specific fence is performed after all previously issued commands within the same tag group and the same command queue. Thus, it is locally ordered with respect to previously issued commands.
b	Tag-specific barrier. A command with a tag-specific barrier is performed after all previously issued commands within the same tag group and the same command queue. Commands that are issued after a command with a tag-specific barrier and that are within the same tag group and the same command queue are also performed after previously issued commands. Thus, commands are locally ordered with respect to previously issued commands. A tag-specific barrier on a command does not affect the order in which that command is performed relative to subsequent commands.
l	List command. Executes a list of list elements located in local storage. The maximum number of elements is 2048. Each element describes a transfer of up to 16 KB.

Commands with an “s” command modifier can only be put into the MFC proxy command queue. Commands with an “l” command modifier and all the MFC atomic commands can only be placed in the MFC SPU command queue. All other commands described in this section can be placed in either of the command queues. Commands issued from a PPE or another processor or device are issued on behalf of an SPE and can be placed in the MFC proxy command queue.

Subsequent sections describe the MFC commands. Each description shows the command mnemonic followed by a list of the mnemonics for the parameters that affect the operation of that command. Not all parameters are used by all commands. As shown in the sample command description that follows, optional parameters, such as MFC effective address high (EAH), are enclosed in brackets. (When EAH is not specified on a command, hardware must set EAH to ‘0’.)

cmd CL, TG, TS/LSZ, LSA, [EAH,] EAL/LA

Table 7-1 lists the parameter mnemonics.

Table 7-1. Parameter Mnemonics

Parameter	Parameter Name	Register Name	See Note
CL	MFC Class ID	MFC_ClassID	
TG	MFC Command Tag Identification	MFC_Tag	
TS	MFC Transfer Size	MFC_Size	1
LSZ	MFC List Size	MFC_Size	1
LSA	MFC Local Storage Address	MFC_LSA	
EAH	MFC Effective Address High	MFC_EAH	2
EAL	MFC Effective Address Low	MFC_EAL	3
LA	MFC List Local Storage Address	MFC_EAL	3
LTS	List Element Transfer Size		4
LEAL	List Element Effective Address Low		4

1. TS and LSZ share the same register offset. The meaning of the contents depends on the command modifier of the MFC opcode.
2. This parameter is optional.
3. EAL and LA share the same register offset. The meaning of the contents depends on the command modifier of the MFC opcode.
4. No associated registers. These parameters are located in local storage and are referenced by the list address (LA) parameter.

Notes:

1. A list command is equivalent to performing a series of commands where the opcode is that of the command without the “l” command modifier. Each element in an MFC list defines one of the series of MFC transfers to perform. For example:

   getl CL, TG, LSZ, LSA, [EAH], LA is equivalent to:

   ```
   while (LSZ > 0)
   {
      get CL, TG, LA(LTS), LSA, [EAH], LA(LEAL)
      /* LSA is moved to next 16-byte boundary in LS */
      LSA = LSA + LA(LTS);
      if (LSA% 16!= 0)
        LSA = (LSA & x‘7FF0’ + 16)
   ```

```
    /* If LSA does not begin on a 16-byte boundary, increment to the next 16-byte boundary.*/
    LA = LA + 8
    LSZ = LSZ - 8
}
```

(For more information about the preceding code, see *Conventions and Notation* on page 17. For more information about list elements, see *Section 7.4 List Commands and List Elements* on page 63.)

2. The effective address (EA) of an MFC command is formed as described below:
   - For nonlist type commands: **(EAH << 32) | EAL**
   - For list type commands: **(EAH << 32) | LA (LEAL**), where **LA (LEAL)** is the effective address low parameter of the current list element pointed to by the list address. This construct assumes the **LA** is incremented after each transfer.
3. The size of a transfer is **TS** bytes for nonlist commands; it is **LA(LTS)** bytes for each element of a list command. For a list command, the number of elements is the MFC List Size (**LSZ)** divided by 8 bytes (that is, 8 bytes per list element).
4. See *Section 7.9 MFC Synchronization Commands* on page 73, for a description of the barrier command modifiers.

7.1 Command Classes

Commands can be categorized into the following three classes:

- Defined
- Illegal
- Reserved

The class of a command is determined by examining the opcode and, if it exists, the extended opcode. If a command opcode, or a combination of an opcode and extended opcode, is not that of a defined or reserved command, then the command is illegal.

A given command is in the same class for all implementations compliant with this release of the Cell Broadband Engine Architecture (CBEA). In future versions of the CBEA, commands that are currently illegal might become defined (by being added to the architecture), or reserved (by being assigned to a special-purpose operation). Similarly, some commands that are currently reserved might become defined in a subsequent architecture release.

7.1.1 Defined Commands

Defined commands fall into one of four categories:

- Data transfer or MFC DMA commands (see *Table 7-2* on page 58)
  – Data moved from local storage and placed in main storage (**put** commands)
  – Data moved into local storage from main storage (**get** commands)
- SL1 storage control commands (see *Table 7-3* on page 59)
- MFC synchronization commands (see *Table 7-4* on page 60)
- MFC atomic commands (see *Table 7-5* on page 60)

The data transfer commands are further divided into subcategories that define the direction of the data movement (that is, to or from local storage). An application can place the data transfer commands listed in *Table 7-2* into the MFC command queues. Unless otherwise noted, these commands can be executed in any order (asynchronously).

Note: Embedded barrier, fence, and synchronization commands *must* be used to ensure proper ordering when ordering is required.

Table 7-2. Data Transfer or MFC DMA Commands (Page 1 of 2)

Mnemonic	Opcode	Support (Proxy/Channel)	Description
Put Commands			
put	x'0020'	Proxy/Channel	Moves data from local storage to an effective address within the main storage domain.
puts	x'0028'	Proxy	Moves data from local storage to an effective address within the main storage domain. Starts the SPU after the DMA operation completes.
putr	x'0030'	Proxy/Channel	Same as **put** with a PPE L2 cache scarf hint (used to send results to a PPE).[1]
putf	x'0022'	Proxy/Channel	Moves data from local storage to an effective address within the main storage domain with fence. This command is locally ordered with respect to all previously issued commands within the same tag group and command queue.
putb	x'0021'	Proxy/Channel	Moves data from local storage to an effective address within the main storage domain 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	x'002A'	Proxy	Moves data from local storage to an effective address within the main storage domain with fence. This command is locally ordered with respect to all previously issued commands within the same tag group and command queue. Starts the SPU after the DMA operation completes.
putbs	x'0029'	Proxy	Moves data from local storage to an effective address within the main storage domain 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. Starts the SPU after DMA operation completes.
putrf	x'0032'	Proxy/Channel	Same as **putf** with a PPE L2 cache scarf hint used to send results to a PPE.[1]
putrb	x'0031'	Proxy/Channel	Same as **putb** with a PPE L2 cache scarf hint used to send results to a PPE. [1]
putl	x'0024'	Channel	Moves data from local storage to an effective address within the main storage domain using an MFC list.
putrl	x'0034'	Channel	Same as **putl** with a PPE L2 cache scarf hint used to send results to a PPE.[1]
putlf	x'0026'	Channel	Moves data from local storage to an effective address within the main storage domain 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	x'0025'	Channel	Moves data from local storage to an effective address within the main storage domain 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.
putrlf	x'0036'	Channel	Same as **putlf** with a PPE L2 cache scarf hint used to send results to a PPE.[1]
putrlb	x'0035'	Channel	Same as **putlb** with a PPE L2 cache scarf hint used to send results to a PPE.[1]

1. Scarfing is the direct transfer of data to a PPE L2 cache.

Table 7-2. Data Transfer or MFC DMA Commands (Page 2 of 2)

Mnemonic	Opcode	Support (Proxy/Channel)	Description
Get Commands			
get	x‘0040’	Proxy/Channel	Moves data from an effective address within the main storage domain to local storage.
gets	x‘0048’	Proxy	Moves data from an effective address within the main storage domain to local storage. Starts the SPU after DMA operation completes.
getf	x‘0042’	Proxy/Channel	Moves data from an effective address within the main storage domain to local storage with fence. This command is locally ordered with respect to all previously issued commands within the same tag group and command queue.
getb	x‘0041’	Proxy/Channel	Moves data from an effective address to local storage 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	x‘004A’	Proxy	Moves data from an effective address within the main storage domain to local storage with fence. This command is locally ordered with respect to all previously issued commands within the same tag group. Starts the SPU after DMA operation completes.
getbs	x‘0049’	Proxy	Moves data from an effective address within the main storage domain to local storage 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. Starts the SPU after DMA operation completes.
getl	x‘0044’	Channel	Moves data from an effective address within the main storage domain to local storage using an MFC list.
getlf	x‘0046’	Channel	Moves data from an effective address within the main storage domain to local storage 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	x‘0045’	Channel	Moves data from an effective address within the main storage domain to local storage 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.

1. Scarfing is the direct transfer of data to a PPE L2 cache.

Table 7-3 lists the storage control commands available for the first level DMA transfer cache (SL1).

Table 7-3. SL1 Storage Control Commands

Mnemonic	Opcode	Support	Description
sdcrt	x'0080'	Proxy/Channel	Brings a range of effective addresses into the SL1 (performance hint for DMA gets).[1]
sdcrtst	x'0081’	Proxy/Channel	Brings a range of effective addresses into the SL1 (performance hint for DMA puts).[1]
sdcrz	x'0089’	Proxy/Channel	Writes zeros to the contents of a range of effective addresses.
sdcrst	x'008D’	Proxy/Channel	Stores the modified contents of a range of effective addresses.
sdcrf	x'008F’	Proxy/Channel	Stores the modified contents of a range of effective addresses and invalidates the block.

1. These commands do not transfer data in implementations without an SL1.

Table 7-4 on page 60 lists the synchronization commands available in the CBEA.

Table 7-4. MFC Synchronization Commands

Command	Opcode	Support	Description
sndsig	x'00A0'	Proxy/Channel	Updates signal notification registers in an I/O device or another SPE. This command is actually a 4-byte DMA put that can go to any address.
sndsigf	x'00A2'	Proxy/Channel	Updates signal notification registers in an I/O device or another SPE with fence. This command is actually a 4-byte DMA put that can go to any address.
sndsigb	x'00A1'	Proxy/Channel	Updates signal notification registers in an I/O device or another SPE with barrier. This command is actually a 4-byte DMA **put** that can go to any address.
barrier	x'00C0'	Proxy/Channel	Barrier type ordering. Ensures ordering of all preceding, nonimmediate DMA commands with respect to all commands following the **barrier** command within the same command queue. The **barrier** command has no effect on the immediate DMA commands: **getllar**, **putllc**, and **putlluc**.
mfceieio	x'00C8'	Proxy/Channel	The **mfceieio** command orders the storage transactions caused by **get** and **put** commands. To ensure that the commands are correctly ordered, the commands must be in the same tag group as the **mfceieio** command, or a barrier command must be issued before the **mfceieio** command. The **mfceieio** command orders transactions as follows, assuming the MFC DMA commands are within the specified tag group. • Orders **get** or **put** commands with respect to other **get** or **put** commands that access storage defined as caching inhibited and guarded. • Orders **put** commands that access storage defined as write through required with respect to **put** or **get** commands that access storage defined as caching inhibited and guarded. • Orders **put** or **get** commands that access storage defined as caching inhibited and guarded with respect to **put** commands that access storage defined as write through required. • Orders **put** commands with respect to **put** commands that access storage that is defined as memory coherency required and is neither write through required nor caching inhibited.
mfcsync	x'00CC'	Proxy/Channel	The **mfcsync** command controls the ordering of DMA **put** and **get** operations within the specified tag group with respect to other processing units and devices in the system.

Table 7-5 lists the atomic commands available in the CBEA.

Table 7-5. MFC Atomic Commands

Command	Opcode	Support	Description
getllar	x'00D0'	Channel	Gets a lock line and creates a reservation (executes immediately).
putllc	x'00B4'	Channel	Puts lock line conditional on a reservation (executes immediately).
putlluc	x'00B0'	Channel	Puts lock line unconditional (executes immediately).
putqlluc	x'00B8'	Channel	Puts lock line unconditional (queued form).

7.1.2 Illegal Commands

The illegal class of commands includes any commands not in the defined class or in the reserved class. Illegal commands are intended for future versions of the CBEA. That is, a future version of the CBEA might define an operation for a command opcode currently in the illegal class.

7.1.3 Reserved Commands

Reserved commands are intended for implementation dependent use. Commands in this class have the reserved upper 8 bits of the MFC command opcode parameter in the range of x‘8000’ ≤ MFC command opcode ≤ x‘FFFF’.

7.2 Command Exceptions

The CBEA does not support unaligned DMAs. If an unaligned DMA operation is encountered, MFC command queue processing is suspended, and a DMA alignment interrupt is generated. The CBEA does not execute invalid DMA commands. If an attempt is made to execute an invalid DMA command, an instruction caused interrupt is generated. *Table 7-6* summarizes the command and alignment errors that the CBEA identifies. For more information, see *Section 21 Interrupt Facilities* beginning on page 261.

Table 7-6. Command Errors and Alignment Errors (Page 1 of 2)

Type of Error	Description of Error
DMA Alignment Errors[1, 2]	
Transfer Size Alignment Error	• The transfer size is not 0, 1, 2, 4, or 8 bytes or a multiple of 16 bytes. • A transfer size is greater than 16 KB. • Any reserved bit is not ‘0’.[3] • The transfer size for a **sndsig** command is not 4 B. **Note:** Requirements for the storage control (SL1) commands are implementation specific. For more information, see the implementation specific documentation.
List Transfer Size Alignment Error	• Any reserved bit is not ‘0’.[3] • A list transfer size is greater than 16 KB.
Local Storage Address Alignment Error	• In the case of a list LS address, LS(0:14) must equal LSA(14:28). The address is aligned on a doubleword (8-byte or 64-bit) boundary. • The following local storage address and transfer size combinations cause an alignment error: – Local storage address bit 31 is not ‘0’ for a transfer size of 2 bytes – Local storage address bits 30 - 31 are not ‘00’ for a transfer size of 4 bytes – Local storage address bits 29 - 31 are not ‘000’ for a transfer size of 8 bytes – Local storage address bits 28 - 31 are not ‘0000’ for a transfer size of a multiple of 16 bytes
Effective Address Alignment Error	• Bits 60 - 63 of the 64-bit effective address formed by EAH ‖ EAL are not equal to LSA(28:31) for all variants of the **put** and **get** commands**,** or the **sndsig** command.
List Address Alignment Error	• Bits 29 - 31 of the list address (LA) are not ‘000’.
DMA Command Errors	
Atomic Command Pending	• A **getllar, putllc,** or **putlluc** is issued to the MFC SPU command queue while another **getllar, putllc,** or **putlluc** is pending.

1. Not checked for **mfcsync**, **mfceieio**, **barrier**, and the MFC atomic commands.
2. An alignment error might not be reported if a command error is present.
3. Some implementations might ignore reserved bits.

Table 7-6. Command Errors and Alignment Errors (Page 2 of 2)

Type of Error	Description of Error
Invalid MFC Command for MFC Command Queue	• An MFC command with an **<l>** modifier is issued to the MFC proxy command queue. • An MFC atomic command is issued to the MFC proxy command queue.
	• An MFC command with an **<s>** modifier is issued to the MFC SPU command queue.
Invalid MFC Command Opcode	• Invalid opcode. For a list of valid opcodes, see *Table 7-2 Data Transfer or MFC DMA Commands* on page 58, *Table 7-3 SL1 Storage Control Commands* on page 59, *Table 7-4 MFC Synchronization Commands* on page 60, and *Table 7-5 MFC Atomic Commands* on page 60. • Any reserved bit is not ‘0’.[3]
Invalid MFC Command Tag	• Any reserved bit is not ‘0’.[3]

1. Not checked for **mfcsync**, **mfceieio**, **barrier**, and the MFC atomic commands.
2. An alignment error might not be reported if a command error is present.
3. Some implementations might ignore reserved bits.

7.3 MFC Command Parameters

The parameters are not the same for the MFC proxy commands (used by PPEs and other devices) and the MFC SPU commands (used by the SPU). The MFC proxy commands do not support the “l” command modifier, while the MFC SPU commands do not support the “s” command modifier. Because the MFC proxy commands do not support the “l” command modifier, they also do not support the LSZ and LA parameters types. In addition, there are issue sequence and policy differences between MFC proxy commands and MFC SPU commands. The issue sequences are described in *Section 8.2 MFC Proxy Command Issue Sequence* beginning on page 88 and in *Section 9.2 MFC SPU Command Issue Sequence* beginning on page 125.

The parameters are supported as described in the following sections:

MFC class ID parameter (***CL***)	See *Section 8.1.2 MFC Class ID Register* on page 82 and *Section 9.1.2 MFC Class ID Channel* on page 118.
MFC command tag identification parameter (***TG***)	See *Section 8.1.3 MFC Command Tag Register* on page 83 and *Section 9.1.3 MFC Command Tag Identification Channel* on page 119.
MFC transfer size parameter (***TS***)	See *Section 8.1.4 MFC Transfer Size Register* on page 84 and *Section 9.1.4 MFC Transfer Size or List Size Channel* on page 120.
MFC list size parameter (***LSZ***)	See *Section 9.1.4 MFC Transfer Size or List Size Channel* on page 120.
MFC local storage address parameter (***LSA***)	See *Section 8.1.5 MFC Local Storage Address Register* on page 85 and *Section 9.1.5 MFC Local Storage Address Channel* on page 121.
MFC effective address high (***EAH***) parameter (optional)	See *Section 8.1.6 MFC Effective Address High Register* on page 86 and *Section 9.1.7 MFC Effective Address High Channel* on page 124.
MFC effective address low (***EAL***) parameter	See *Section 8.1.7 MFC Effective Address Low Register* on page 87 and *Section 9.1.6 MFC Effective Address Low or List Address Channel* on page 122.
MFC list local storage address (***LA***) parameter	See *Section 9.1.6 MFC Effective Address Low or List Address Channel* on page 122.

7.4 List Commands and List Elements

Commands with an “l” command modifier are list commands. They use list elements located in the local storage pointed to by the MFC list local storage address (LA) parameter of a list command. The element contains the lower order word of the effective address (LEAL) and the list element transfer size (LTS). The element also contains a stall-and-notify flag.

The list commands use a list of effective addresses and transfer size pairs, or list elements, stored in local storage as the parameters for the DMA transfer. The first word contains the transfer size and a stall-and-notify flag. The second word contains the lower order 32 bits of the effective address. While the starting effective address is specified for each transfer element in the list, the local storage address involved in the transfer is only specified in the primary list command. The local storage address is internally incremented based on the amount of data transferred by each element in the list.

List commands are not supported on the MFC proxy command queue.

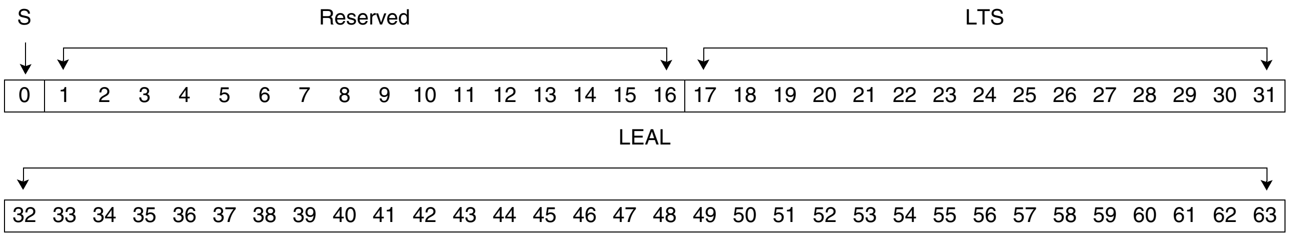
Special handling is performed for list elements with a transfer size less than 16 bytes. In this case, the local storage address for the transfer is adjusted to have the same quadword (16-bytes) alignment as the effective address for the transfer. Following the transfer, the local storage address is internally incremented by the amount of data transferred. If this address does not begin on a 16-byte boundary for a list element transfer, the hardware automatically increments the local storage address to the next 16-byte boundary.

Effective addresses specified in the list elements are relative to the 4 GB area defined by the upper 32 bits of the effective address specified in the base list command. While MFC list starting addresses are relative to the single 4 GB area, transfers within a list element can cross the 4 GB boundary.

Setting the Stall-and-Notify (S) bit causes the DMA operation to suspend execution of this list after the current list element has been processed and to set a stall-and-notify event status for the SPU. Execution of the stalled list does not resume until the MFC receives a stall-and-notify acknowledgment from the SPU program. Stall-and-notify events are posted to the SPU program using the associated command tag group identifier. When there are multiple list commands in the same tag group with stall-and-notify elements, software should ensure that a tag-specific **barrier** or global **barrier** forces ordered execution of the list commands to avoid ambiguity. Setting the S bit on the last element in a list is not supported and will be ignored.

All list elements within the list command are guaranteed to be started and issued in sequence. All elements within a list command have an inherent local ordering. All transfers to guarded pages within an MFC list must adhere to the caching inhibited and guarded page semantics with respect to ordering.

A single list command can contain up to 2048 elements, occupying 16 KB of local storage.



Bits	Field Name	Description
0	S	Stall-and-notify bit
1:16	Reserved	Reserved
17:31	LTS	List element transfer size (LTS)

Bits	Field Name	Description
32:63	LEAL	Low word of the 64-bit effective address (LEAL)

Programming Note:

If at all possible, list elements with transfer sizes of 128 bytes or more should be aligned on 128-byte boundaries for maximum performance. Smaller transfers and transfers not aligned on 128-byte boundaries result in poorer performance. Transfers of less than 16 bytes are inefficient and should be used only when necessary to interface with I/O devices.

7.5 Get Commands (Main Storage to Local Storage)

Get commands are conventional DMA transfer commands. The memory management unit (MMU) translates the effective address provided with the MFC get command into a real address as appropriate. Then, MFC get commands copy the number of bytes of storage specified by the transfer size parameter from the translated effective address (that is, the real address) to the destination local storage address.

When the **get** and **get<f,b>** commands have an “s” command modifier, they can set the Run bit after the local storage has been updated. Commands with an “s” command modifier can only be executed from the MFC proxy command queue.

7.5.1 Get Command

The get (**get*[s]***) command transfers the number of bytes specified by the transfer size parameter from the effective address to the local storage address of the corresponding SPU.

get	***CL, TG, TS, LSA, [EAH,] EAL***
gets	***CL, TG, TS, LSA, [EAH,] EAL***

7.5.2 Get with Fence or with Barrier Command

Like the **get** command, the get with fence or with barrier (**get*<f,b>[s]***) command transfers the number of bytes specified by the transfer size parameter from the effective address to the local storage address of the corresponding SPU. Unlike the **get** command, the **get*<f,b>*** command provides local ordering with respect to other commands within the same tag group (that is, commands with the same tag ID).

getf	***CL, TG, TS, LSA, [EAH,] EAL***
getb	***CL, TG, TS, LSA, [EAH,] EAL***
getfs	***CL, TG, TS, LSA, [EAH,] EAL***
getbs	***CL, TG, TS, LSA, [EAH,] EAL***

7.5.3 Get List Command

The get list (**getl**) command lets software transfer discontinuous blocks of data from the effective address space to a contiguous area of local storage using a single DMA list command. This command uses a list of list elements; each element consists of an effective address (low) and a transfer size pair. This list, the MFC list, is stored in local storage and is the source of the DMA operation. The list address parameter of this command contains the starting address of the MFC list in local storage. The list address is a local storage address, and is therefore not translated by the MMU. The number of bytes in the list is provided in the list size parameter. The list size parameter must be a multiple of 8 bytes for this DMA command, and the list address parameter must be aligned on an 8-byte boundary in local storage. The maximum list size is architecturally 16 KB. However, the supported list size is implementation dependent. This command is not available from the MFC proxy command queue.

The DMA parameters provided in local storage must follow the same format as the **get** command. The local storage address (LSA) must start on a 16-byte boundary unless the transfer size of the first list element is less than 16 bytes.

The effective address (high) parameter is provided as part of the **getl** command. While the starting effective address is always EAH || LEAL, a DMA transfer within a list element can cross the 2^{32} boundary.

getl ***CL, TG, LSZ, LSA, [EAH,] LA***

Note: A DMA command with list operation that uses a local storage destination within the list area produces unpredictable results, if the transfer modifies list elements not yet started.

7.5.4 Get List with Fence or with Barrier Command

Like the **getl** command, the get list with fence or with barrier (**getl*<f,b>***) command transfers discontinuous blocks of data from the effective address space to a contiguous area of local storage. Unlike the **getl** command, the **getl*<f,b>*** command provides local ordering with respect to other commands within the same tag group (that is, commands with the same tag ID). This command is not available from the MFC proxy command queue.

getlf ***CL, TG, LSZ, LSA, [EAH,] LA***
getlb ***CL, TG, LSZ, LSA, [EAH,] LA***

7.6 Put Commands (Local Storage to Main Storage)

Put commands are conventional DMA transfers. The MMU translates the effective address provided with the MFC **put** command into a real address as appropriate. Then, MFC **put** commands transfer the number of bytes specified by the transfer size parameter from the source local storage address to the translated effective address (that is, the real address).

When the **put** and **put*<f,b>*** commands have an “s” command modifier, they can set the Run bit after the DMA operation completes. Commands with an “s” command modifier can only be executed from the MFC proxy command queue.

7.6.1 Put Command

The put (**put*[s]***) command transfers the number of bytes specified by the transfer size parameter from the local storage address of the corresponding SPU to the effective address.

put ***CL, TG, TS, LSA, [EAH,] EAL***
puts ***CL, TG, TS, LSA, [EAH,] EAL***

7.6.2 Put with Fence or with Barrier Command

Like the **put** command, the put with fence or with barrier (**put*<f,b>[s]***) command transfers the number of bytes specified by the transfer size parameter from the local storage address of the corresponding SPU to the effective address. Unlike the **put** command, the **put*<f,b>*** command provides local ordering with respect to other commands within the same tag group (that is, commands with the same tag ID).

putf	***CL, TG, TS, LSA, [EAH,] EAL***
putb	***CL, TG, TS, LSA, [EAH,] EAL***
putfs	***CL, TG, TS, LSA, [EAH,] EAL***
putbs	***CL, TG, TS, LSA, [EAH,] EAL***

7.6.3 Put List Command

The put list (**putl**) command lets software transfer a contiguous area of local storage to discontinuous areas in the effective address space using a single DMA list command. This command uses a list of list elements; each element consists of an effective address (low) and a transfer size pair. This list, the MFC list, is stored in local storage and is the destination of the DMA operation. The list address parameter of this command contains the starting address of the MFC list in local storage. The list address is a local storage address, and is therefore not translated by the MMU. The number of bytes in the list is provided in the list size parameter. The list size parameter must be a multiple of 8 bytes for this DMA command, and the list address parameter must be aligned on an 8-byte boundary in local storage. The maximum list size is architecturally 16 KB. However, the supported list size is implementation dependent. This command is not available from the MFC proxy command queue.

The DMA parameters provided in local storage must follow the same format as the **put** command. The local storage address (LSA) must start on a 16-byte boundary unless the transfer size of the first list element is less than 16 bytes.

The effective address (high) parameter is provided as part of the **putl** command. While the starting effective address is always EAH || LEAL, a DMA transfer within a list element can cross the 2^{32} boundary.

putl	***CL, TG, LSZ, LSA, [EAH,] LA***

7.6.4 Put List with Fence or with Barrier Command

Like the **putl** command, the put list with fence or with barrier (**putl*<f,b>***) command transfers data from a contiguous area of local storage to discontinuous blocks in the effective address space. Unlike the **putl** command, the **putl*<f,b>*** command provides local ordering with respect to other commands within the same tag group (that is, commands with the same tag ID). This command is not available from the MFC proxy command queue.

putlf	***CL, TG, LSZ, [EAH,] LA***
putlb	***CL, TG, LSZ*, *LSA, [EAH,] LA***

7.6.5 Put Result (hint) Command

The **put**, **putl**, **put*<f,b>***, and **putl*<f,b>*** commands also support a result modifier (**putr**, **putr*<f,b>***, **putrl**, **putrl*<f,b>***). These commands perform the same operations. However, they also provide a hint to the MFC that the data for the transfer can be directly transferred into a PPE L2 cache. This is only a hint for performance. If the specified cache line is not found in a PPE L2 cache, the data will be written to system memory. This form of these commands enables an SPU to deliver results directly to a PPE by updating the PPE L2 cache.

Implementation Note:

The current implementation for the CBEA-compliant processor does not support the direct transfer of data to the PPE L2 cache (called scarfing). These commands are treated as if the result hint were not provided.

7.7 Storage Control Commands

The storage control commands described in this section are similar to the PowerPC storage control instructions defined in *PowerPC Architecture, Book II*. SL1 storage control commands affect the number of bytes of storage specified by the transfer size parameter. The description of each command explains how it affects storage. These storage control commands all create a tag-specific barrier even though there is no **<b>** modifier. Thus, the initial storage control command and all subsequent commands in the same tag group are ordered with respect to all previous commands in the queue that have the same tag group ID. Subsequent commands in the same tag group as the SL1 storage control command will not start until the storage control command has completed.

7.7.1 SL1 Data Cache Range Touch Command

The SL1 data cache range touch (**sdcrt**) command is a hint to the MFC that the SPU will probably issue a DMA **get** command to the range of addresses specified by the effective address and transfer size parameters. This command only provides a hint to the MFC; it does not cause any data transfers. Execution of the **sdcrt** command does not cause the system error handler to be invoked. Updates of the Reference and Change (RC) bits within the page table entry (PTE) are not required for this operation.

sdcrt ***CL, TG, TS, [EAH,] EAL***

Implementation Note:

The MFC responds to the hint provided by the **sdcrt** command by taking actions to reduce the latency of subsequent MFC DMA **get** and **put** commands that access the specified block. (Such actions might include prefetching the block into levels of the storage hierarchy that are near an SPE.) For example, the MFC might prefetch data into an SL1 that is associated with the SPU that issued the **sdcrt** command. Doing so affects the state of other system caches, including other atomic update facilities, as required by the coherency protocol of the system.

An SPE might implement an atomic update facility independent from the SL1 for the reservations created by the **getllar** command. If so, the MFC **sdcrt** command has no effect on any data cached in this facility.

7.7.2 SL1 Data Cache Range Touch for Store Command

The SL1 data cache range touch for store (**sdcrtst**) command is a hint to the MFC that the SPU will probably issue a DMA **put** command to the range of addresses specified by the effective address and transfer size parameters. This command only provides a hint to the MFC; it does not cause any data transfers. Execution of the **sdcrtst** command does not cause the system error handler to be invoked. Updates of the Reference and Change (RC) bits within the PTE are not required for this operation.

sdcrtst ***CL, TG, TS, [EAH,] EAL***

Implementation Note:

The MFC responds to the hint provided by the **sdcrtst** command by taking actions to reduce the latency of subsequent MFC DMA **get** and **put** commands that access the specified block. (Such actions might include prefetching the block into levels of the storage hierarchy that are near an SPE.) For example, the MFC might prefetch data into an SL1 that is associated with the SPU that issued the **sdcrtst** command using a protocol that tells the system that the data is intended to be modified. Doing so affects the state of other system caches, including other atomic update facilities, as required by the coherency protocol of the system.

An SPE might implement an atomic update facility independent from the SL1 for the reservations created by the **getllar** command. If so, the MFC **sdcrtst** command has no effect on any data cached in this facility.

7.7.3 SL1 Data Cache Range Set to Zero Command

The SL1 data cache range set to zero (**sdcrz**) command sets the range of storage specified by the effective address and transfer size parameters to zero. This command does not cause the data to exist in the SL1 if the storage area is caching inhibited.

sdcrz ***CL, TG, TS, [EAH,] EAL***

Note: All the bytes in the data block containing the byte addressed by the effective address are set to 0. Doing so affects the state of other system caches, including the SL1s and other atomic update facilities, as required by the coherency protocol of the system.

Implementation Note:

An SPE might implement an atomic update facility independent from the SL1 for the reservations created by the **getllar** command. If so, the MFC **sdcrz** command does not cause data to be brought into the atomic facility if the data block is not already present in this facility.

7.7.4 SL1 Data Cache Range Store Command

The SL1 data cache range store (**sdcrst**) command causes a data block in the associated SL1 cache and any other processor cache to be written to main storage when it meets the following conditions:

- The data block is in memory-coherence-required storage.
- The data block is considered modified.
- The data block is within the address range defined by the effective address and transfer size parameters.

The data block can remain in the processor cache, but it is no longer considered modified.

If a data block in the associated SL1 cache is *not* in memory-coherence-required storage and is considered modified, the **sdcrst** command writes the modified block of only the SL1 cache associated with the issuing SPU to main storage. The data block can remain in the cache, but it is no longer considered modified.

Updates of R and C bits of the PTE are not required for this operation.

sdcrst ***CL, TG, TS, [EAH,] EAL***

Note: All the bytes in the modified data block that contains the byte addressed by the effective address are written to main storage, even if only one byte is modified. Doing so affects the state of other system caches, including the SL1s and other atomic update facilities, as required by the coherency protocol of the system.

Implementation Note:

An SPE might implement an atomic update facility independent from the SL1 for the reservations created by the **getllar** command. If so, the MFC **sdcrtst** command can affect the data present in this facility. If any byte in the effective address range specified by this command is present and considered modified, the modified data block is written to main storage.

7.7.5 SL1 Data Cache Range Flush Command

The SL1 data cache range flush (**sdcrf**) command causes a data block in the associated SL1 cache and any other processor cache to be written to main storage when it meets the following conditions:

- The data block is in memory-coherence-required storage.
- The data block is considered modified.
- The data block is within the address range defined by the effective address and transfer size parameters.

All data blocks in the effective address range are invalidated.

If a data block in the associated SL1 cache is *not* in memory-coherence-required storage and is considered modified, the **sdcrf** command writes the modified block of only the SL1 cache associated with the issuing SPU to main storage. All data blocks in the effective address range are invalidated.

Updates of R and C bits of the PTE are not required for this operation.

sdcrf ***CL, TG, TS, [EAH,] EAL***

Note: All the bytes in the modified data block that contains the byte addressed by the effective address are written to system memory. Doing so affects the state of other system caches, including the SL1s and other atomic update facilities, as required by the coherency protocol of the system.

Implementation Note:

An SPE might implement an atomic update facility independent from the SL1 for the reservations created by the **getllar** command. If so, the MFC **sdcrf** command can affect the data present in this facility. If any byte in the effective address range specified by this command is present and considered modified, the modified data block is written to main storage and the data block is invalidated. If any byte is present and not considered modified, the data block is also invalidated. Since the data block has not been modified, it does not need to be written to main storage.

7.8 MFC Atomic Update Commands

There are four MFC atomic update commands: **getllar**, **putllc**, **putlluc**, and **putqlluc.** The **getllar**, **putllc**, and **putlluc** MFC atomic update commands are performed without waiting for other commands in the MFC SPU queue and thus have no associated tag. The MFC performs the atomic update operations independently of any pending **mfcsync**, **mfceieio**, or **barrier** commands in the MFC SPU command queue. Software must issue a read (**rdch**) from the MFC Read Atomic Command Status Channel (see page 137) after issuing each atomic update command to verify completion of the command.

The MFC commands for get lock line and reserve (**getllar**) and put lock line conditional (**putllc**) are similar to the PowerPC **lwarx**, **ldarx**, **stwcx**, and **stdcx** instructions for use in atomic updates. For interoperability between the PowerPC instructions and the SPE commands, the PPE and SPE must have the same reservation granule size. A reservation granule is the number of bytes for which the reservation is held. The put lock line unconditional (**putlluc**) command and the put queued lock line unconditional (**putqlluc**) command perform a similar function to a cacheable store instruction in the PowerPC Architecture, conventionally used by software to release a lock. The difference between the **putlluc** and **putqlluc** commands is that the **putqlluc** command is tagged and queued behind other MFC commands in the MFC SPU command queue, whereas the **putlluc** command is executed immediately. Because the **putqlluc** command is tagged and creates a tag-specific fence, it is ordered with respect to all other commands in the same tag group already in the MFC SPU command queue.

The **getllar**, **putllc**, and **putlluc** commands are executed immediately. However, these commands still require an available slot in the MFC SPU command queue. No ordering with other commands in the MFC SPU command queue should be assumed. After issuing each **getllar**, **putllc**, or **putlluc** command, software must issue a read from the MFC Read Atomic Command Status Channel (see page 137) to verify completion of the command.

All MFC atomic update commands must be issued to memory where the storage attributes are neither write through required nor caching inhibited (W = ‘0’ and I = ‘0’). Issuing a **getllar**, **putllc**, **putlluc**, or **putqlluc** command to memory where the storage attributes are write through required or caching inhibited (W = ‘1’ or I = ‘1’) is undefined. This can cause the interrupt handler for a class 1 MFC interrupt to be invoked.

See the *PowerPC Architecture, Book II* for examples of the use of atomic updates. The MFC atomic update commands can only be placed in the MFC SPU command queue by following the sequence described in *Section 9.2 MFC SPU Command Issue Sequence* beginning on page 125.

Programming Note:

The storage attribute of memory coherence required does not have to be specified for memory that is accessed by atomic operations. However, specifying the storage attribute of memory coherency required (M = ‘1’) ensures that stores or puts from other PPEs, SPEs, or devices cause the reservation created by a **getllar** command to be lost.

7.8.1 Get Lock Line and Reserve Command

A get lock line and reserve (**getllar**) command is similar to the PowerPC **lwarx** and **ldarx** instructions except for the size of the data transfer and the destination of the data. The data transfer size of the **getllar** command is a reservation granule (that is, an aligned unit of real storage). The data for the **getllar** command is placed in local storage; the data for a **lwarx** or **ldarx** instruction is placed in a general purpose register (GPR).

The **getllar** ***CL, LSA, [EAH,] EAL*** command cannot be issued from the MFC proxy command queue.

Issuing the **getllar** command requires an available MFC SPU command queue slot. However, this command is issued immediately, is not queued behind other commands, and has no associated tag. Therefore, it executes independently of pending **mfcsync**, **mfceieio**, or **barrier** commands in the queue. An attempt to issue this command before a previous **getllar**, **putllc**, or **putlluc** command has completed results in an error (MFC command queue processing is halted, and an interrupt is sent to a PPE).

A read channel (**rdch**) of the MFC Read Atomic Command Status Channel (see page 137) must be performed after issuing this command and before issuing another **getllar**, **putllc,** or **putlluc** command.

Privileged software must issue a **putllc** command to a privileged address to reset the reservation of an application as part of an SPE context switch. Issuing a **putllc** command resets the reservation if a context switch occurred after the **getllar** made a reservation but before the **putllc** used the reservation. This prevents the resumed context from inadvertently using the reservation of the previous context if it was switched at a similar point using the same address.

Software must avoid issuing successive **getllar** commands with the same effective address (having the same reservation granule) without an intervening **putllc** command unless a random backoff technique is used to avoid livelock situations. (A livelock is an endless loop in program execution.) A random backoff technique provides a random amount of delay between issues of two or more successive **getllar** commands. A livelock can occur during locking or barrier sequences such as test and set, compare and swap, or repeatedly polling for a value to change, using the **getllar** command before performing an atomic update or another action.

Programming Note:

Avoid looping on **getllar** commands. This helps to prevent the starvation of writes or low-priority reads that can result from back-to-back high-priority reads taking precedence in the arbitration scheme.

When using the atomic update sequence in a **barrier** or synchronization operation (such as compare and swap or test and set), consider using the Lock Line Reservation Lost Event (see page 170) instead of repeatedly issuing the **getllar** command. Using 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, the programmer can perform a read channel (**rdch**) from the SPU Read Event Status Channel (see page 153) to put the SPU into low power state until the lock line data has been modified.

Note: When a reservation exists in the atomic unit, issuing another **getllar** command can cause the existing reservation to be lost. The reservation is lost if the effective address of the **getllar** command is not within the reservation granule of the existing reservation (that is, the effective address does not address a byte within the group of bytes for the existing reservation). This can cause a reservation lost event for the previous reservation. In addition, a context switch can cause a reservation lost event. A reservation lost event only specifies that data within the reservation granule might have been modified.

7.8.2 Put Lock Line Conditional Command

A put lock line conditional (**putllc**) command is similar to the PowerPC **stwcx.** and **stdcx.** instructions except for the size of the store conditional and the source of the data. The data transfer size of the **putllc** command is a cache line. The data for the **putllc** command is read from the local storage; the data for an **stwcx.** or **stdcx.** instruction is read from a GPR.

The **putllc** ***CL, LSA, [EAH,] EAL*** command cannot be issued from the MFC proxy command queue.

Issuing the **putllc** command requires an available MFC SPU command queue slot. However, this command is issued immediately, is not queued behind other commands, and has no associated tag. Therefore, this command executes independently of pending **mfcsync**, **mfceieio**, or **barrier** commands in the queue. An attempt to issue this command before a previous **getllar**, **putllc**, or **putlluc** command has completed results in an error (MFC command queue processing is halted, and an interrupt is sent to a PPE).

The **putllc** command is a conditional store operation. The store is not successful if no reservation for the same address has been made, or if the reservation has been lost. A read channel (**rdch**) of the MFC Read Atomic Command Status Channel (see page 137) is required to verify the completion of this command. An SPU indefinite stall will result if the MFC Read Atomic Command Status Channel is read without the issuance of a lock line MFC command.

7.8.3 Put Lock Line Unconditional Command

A put lock line unconditional (**putlluc**) command is similar to the **putllc** command, except that the store for the **putlluc** is always performed. The **putlluc** command store is not dependent upon the existence of a previously made reservation. The data transfer size of the **putlluc** command is a cache line.

The **putlluc** ***CL, LSA, [EAH,] EAL*** command cannot be issued from the MFC proxy command queue.

Issuing the **putlluc** command requires an available MFC SPU command queue slot. However, this command is issued immediately, is not queued behind other commands, and has no associated tag. Therefore, it executes independently of pending **mfcsync**, **mfceieio**, or **barrier** commands in the queue. An attempt to issue this command before a previous **getllar**, **putllc**, or **putlluc** command completes results in an error (MFC command queue processing is halted, and the PPE is interrupted).

The **putlluc** command is a store operation. The store is not conditional on having acquired a reservation. A read channel (**rdch**) of the MFC Read Atomic Command Status Channel (see page 137) is required to clear the status of the operation. A read of the MFC Read Atomic Command Status Channel is also required to verify the completion of this command. An SPU indefinite stall will result if the MFC Read Atomic Command Status Channel is read without the issuance of a put lock line DMA command.

7.8.4 Put Queued Lock Line Unconditional Command

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** command is performed immediately and the **putqlluc** command is placed into the MFC SPU command queue along with other MFC commands.

The **putqlluc** ***CL, TG, LSA, [EAH,] EAL*** command cannot be issued from MFC proxy command queue.

No ordering is performed between the queued **putqlluc** command and the immediate **getllar**, **putllc**, or **putlluc** commands. Thus, a **putqlluc** can execute before or after a subsequently issued or a previously issued immediate atomic command (**getllar**, **putllc**, or **putlluc**). Therefore, do not assume any order of execution with respect to immediate lock-line 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 tag-group completion. See *Section 9.3 MFC Tag-Group Status Channels* beginning on page 126 for more information.

The **putqlluc** command contains a tag parameter and creates a tag-specific fence even though there is no **<f>** modifier. The tag-specific fence created by the **putqlluc** command prevents this command from being issued until all previously issued commands with the same tag have completed. Unlike the immediate form of this command, multiple **putqlluc** commands can be issued and pending in the DMA command queue. All the MFC command queue ordering rules apply to the **putqlluc** command.

This command is a store operation. The store is not conditional on having acquired a reservation.

Programming Note:

The put queued lock line unconditional (**putqlluc**) command allows software to queue the release of a lock behind the commands accessing storage associated with the lock. For proper operation, the **putqlluc** command must be within the same tag group as the commands accessing the associated storage or other ordering commands must be used. In addition, either **mfceieio** or **mfcsync** commands must also be used (as appropriate).

7.9 MFC Synchronization Commands

MFC synchronization commands control the order in which storage accesses are performed with respect to other MFCs, processors, and other devices.

Many commands support the embedded tag-specific fence modifier, or barrier modifier. The notation <***f,b***> indicates that either the tag-specific fence or tag-specific barrier form is available. The tag-specific fence, **<f>**, ensures that this command is ordered with respect to all preceding commands in the DMA command queue within the same tag group. Any subsequent command with the same tag ID that is not a fence or barrier, or any command within a different tag group, can be performed out-of-order with respect to this command or previously-issued commands. The tag-specific barrier, <***b***>, ensures that this command and all subsequent commands within the same tag group as this command are ordered with respect to all preceding commands in the DMA command queue within the same tag group. This command and all subsequent commands with the same tag ID as this command will not be performed out-of-order with respect to all preceding commands with the same tag ID as this command. Once all previously-issued commands with the same tag ID as this command have been performed, this command and subsequent commands can be performed. The order in which these subsequent commands are performed is determined by subsequent fence and barrier form commands. Commands with a fence or barrier are not ordered with respect to subsequent commands.

The fence and barrier modifiers also provide stronger consistency of storage accesses in the weakly consistent storage model of the CBEA for several combinations of storage accesses involving commands in the same tag group. Thus, programmers do not need to add additional synchronization commands for these specific combinations if the commands are in the same tag group and either the fence or barrier modifier is used. The following DMA combinations require no additional synchronization commands to provide an ordering function when both commands access storage that has the same storage attributes:

- Same tag group **put(l)** type command or **sdcrz** command followed by a **put(l)** type command with either the fence **<f>** or the barrier **<b>** modifier or by an **sdcrz** command
- Same tag group **put(l)** type command or **sdcrz** command followed by a **get(l)** type command with either the fence **<f>** or the barrier **<b>** modifier
- Same tag group **get(l)** type command followed by a **get(l)** type command with either the fence **<f>** or the barrier **<b>** modifier

- Same tag group **get(l)** type command followed by a **put(l)** type command with either the fence **<f>** or the barrier **<b>** modifier or by an **sdcrz** command

If the storage accesses between two MFC DMA commands that access storage with different storage attributes need to be strongly ordered with respect to other processors, MFCs, and devices, an **mfcsync** or **mfceieio** command *must* be issued between the commands to provide the required ordering function. In addition, the two MFC DMA commands and the **mfcsync** command must all be in the same tag group. For more information on the use of **mfcsync** and **mfceieio**, see *Section 10.2 Main Storage Domain Access Ordering* on page 177.

Implementation Note:

The CBEA specifies that the fence and barrier modifiers provide stronger storage access consistency for some combinations of DMA commands. The following items explain how the ordering rules for DMA command storage access relate to the PowerPC ordering rules:

- A **put(l)** type command or an **sdcrz** command followed by a **put(l)** type command with either the fence **<f>** or barrier **<b>** modifier or by an **sdcrz** command, 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 a PPE separated by a **lwsync** instruction.
- A **get(l)** type command followed by a **get(l)** type command with either the fence **<f>** or barrier **<b>** modifier, 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 a PPE separated by a **lwsync** instruction.
- A **put(l)** type command or an **sdcrz** command followed by a **put(l)** type command with either the fence **<f>** or barrier **<b>** modifier or by an **sdcrz** command, where the storage attributes are caching inhibited, has a storage order effect at least as strong as two stores on a PPE separated by a **sync** instruction.
- A **get(l)** type command followed by a **get(l)** type command with either the fence **<f>** or barrier **<b>** modifier, where the storage attributes are caching inhibited and not guarded, has a storage order effect at least as strong as two loads on a PPE separated by a **sync** instruction.
- A **get(l)** type command followed by a **get(l)** type command with either the fence **<f>** or barrier **<b>** modifier, where the storage attributes are caching inhibited and guarded, has a storage order effect at least as strong as two loads on a PPE separated by an **eieio** instruction.
- A **put(l)** type command or an **sdcrz** command followed by a **get(l)** type command with either the fence **<f>** or barrier **<b>** modifier, 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 a PPE separated by a **sync** instruction.
- A **get(l)** type command followed by a **put(l)** type command with either the fence **<f>** or barrier **<b>** modifier or by an **sdcrz** command, 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 a PPE separated by a **lwsync** instruction.
- A **get(l)** type command followed by a **put(l)** type command with either the fence **<f>** or barrier **<b>** modifier or by an **sdcrz** command, 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 a PPE separated by a **sync** instruction.
- A **get(l)** type command followed by a **put(l)** type command with either the fence **<f>** or barrier **<b>** modifier or by an **sdcrz** command, where the storage attributes are caching inhibited and guarded, has a stor-

age order effect at least as strong as a load followed by a store on a PPE separated by an **eieio** instruction.

- A **put(l)** type command or an **sdcrz** command followed by a **put(l)** type command with either the fence **<f>** or barrier **<b>** modifier or by an **sdcrz** command, where the storage attributes are caching inhibited and guarded, are always performed in program order with respect to any processor or mechanism.

An example of the use of a fence modifier is when multiple MFC DMA commands are needed to load an SPU program and to start its execution. In this example, one MFC DMA command is used to load the data segment. A second MFC DMA command with both the SPU start “s” and the tag-specific fence <**f**> modifiers is used to load text data. As long as 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 MFC DMA command could complete and could start the SPU program before the data segment is loaded.

An example of the use of a barrier modifier is when a read of a buffer 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 modifier for the first command to write the buffer allows the commands used to write the buffer to be queued without waiting for the MFC tag-group status update event on the read commands. (The barrier modifier is only required for the first buffer write command.) As long as 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. This provides better performance but forces all reads to finish before the writes start.

7.9.1 MFC Synchronize Command

An MFC synchronize (**mfcsync**) command is similar in operation to the PowerPC **sync** instruction.

mfcsync ***TG***

This command is used to control the order in which DMA commands within this tag group are executed with respect to other processors and devices to the extent required by the associated memory-coherence-required attributes.[1] The **mfcsync** command creates a tag-specific barrier even though there is no **<b>** modifier. This means that 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. The ordering done by the **mfcsync** command for main storage accesses is cumulative in the main storage domain. However, the ordering is not cumulative with respect to other local storage domains. The MFC Multisource Synchronization Register (see page 109) defines a facility to achieve cumulative ordering across the other local storage and main storage domains.

For main storage with certain storage attributes, the **mfceieio** command is used to control the order in which DMA commands within the same tag group are executed with respect to other processors and devices. These main storage accesses are divided into the following two sets, which are ordered separately:

1. The phrase “to the extent required by the associated memory-coherence-required attributes” refers to the memory-coherence-required attribute, if any, associated with each main storage access. This phrase does not apply to storage accesses in the local storage domain.

1. The **mfceieio** command orders **put(l)**, **sdcrz**, and **get(l)** commands to main storage that is both caching inhibited and guarded. It also orders **get(l)** commands to main storage that is write through required. The ordering done by the **mfceieio** command for main storage accesses in this set is not cumulative.
2. The **mfceieio** command orders **put(l)** or **sdcrz** commands to main storage that is memory coherence required and is neither write through required nor caching inhibited. The ordering done by the **mfceieio** command for main storage accesses in this set is cumulative in the main storage domain. However, the ordering is not cumulative with respect to other local storage domains. The MFC Multisource Synchronization Register (see page 109) defines a facility to achieve cumulative ordering across the other local storage and main storage domains.

Programming Note:

To obtain the best performance across the widest range of implementations, the programmer should use either the **barrier** command, the fence **<f>** or barrier **<b>** modifiers, or the **mfceieio** command if any of these are sufficient for the ordering needs.

7.9.2 MFC Enforce In Order Execution of I/O Command

An MFC enforce in-order execution of I/O (**mfceieio**) command is similar in operation to the PowerPC **eieio** instruction.

mfceieio ***TG***

This command is used to control the order in which DMA commands within this tag group are executed with respect to other processors and devices to the extent required by the associated memory-coherence-required attributes.[1] The **mfceieio** command creates a tag-specific barrier even though there is no **<b>** modifier. This means that the **mfceieio** 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.

Programming Note:

This command is intended for use in managing shared data structures, in performing memory-mapped I/O, and in preventing load and store combining in system memory. To obtain the best performance across the widest range of implementations, the programmer should use either a fence or barrier form of a command, or if neither of these is sufficient for ordering, the **mfceieio** command.

7.9.3 Barrier Command

A **barrier** command orders all subsequent commands with respect to all commands preceding the **barrier** command in the DMA command queue, independent of tag group IDs. The **get*<f,b>***, **getl*<f,b>***, **put*<f,b>***, and **putl*<f,b>*** commands have an embedded form of fence or barrier that only affects commands within the same tag group.

barrier ***TG***

1. The phrase "to the extent required by the associated memory-coherence-required attributes" refers to the memory-coherence-required attribute, if any, associated with each main storage access. This phrase does not apply to storage accesses in the local storage domain.

The **barrier** command is not tag-specific. The specified tag can be used to determine when the command is complete. The **barrier** command does not complete until all preceding commands in the queue are complete. Subsequent commands in the queue begin when the **barrier** command completes.

The barrier command has the same storage access ordering properties independent of the tag group as the embedded **fence** or **barrier** modifier for commands in the same tag group.

7.9.4 Send Signal Command

A send signal (**sndsig**) command logically sets signal bits in the targeted signal notification register. The targeted signal notification register is specified by the effective address. The data used to update the signal notification register is the data from the location specified by the local storage address.

This operation is always a 4-byte operation, and the transfer size (***TS***) parameter must be set to four.

sndsig ***CL, TG, TS, LSA, [EAH,] EAL***

The **sndsig** command is normally used to signal events between SPEs, between PPEs and SPEs, or between SPEs and I/O devices. Each SPE has two signal notification registers that can be targeted by this command. A PPE has no signal control registers. It can only initiate signals by using the **sndsig** command. I/O devices can initiate signals by using the **sndsig** command; optionally, I/O devices can also have signal control registers.

The signal notification registers can be programmed for overwrite mode or logical OR mode. For more information, see *Section 16.4 SPU Configuration Register* beginning on page 245. In overwrite mode, the contents of the signal notification register are replaced with the signal information set by this command. This mode is useful in a one-to-one signalling environment. In the logical OR mode, the contents of the signal notification register are logically ORed with the signal information set by this command. This mode is useful in a many-to-one signalling environment.

A processor that overwrites the signal notification register data does not have hardware protection. When the signal control register is read locally by the signal control owner, any signal bits that are set are reset. Any remote (nonowner) read of these signal control registers returns the current state of the signal control register but does not result in a reset. SPUs read and reset signal control registers through SPU channels.

MMIO addresses are provided for PPEs or I/O devices to issue the **sndsig** command to a specified signal notification register. The PPEs can also set or clear the signal control registers of SPEs to establish the context through the SPU channel access facility.

An SPE that receives a **sndsig** operation must guarantee that all store operations sent to the local storage from a single source are complete before depending on the data associated with the **sndsig** operation. The MFC multisource synchronization facility (see page 108) must be used to ensure that stores from multiple sources are complete.

Implementation Note:

The **sndsig** command can be implemented as a **put** command.

7.9.5 Send Signal with Fence or with Barrier Command

The send signal with fence or with barrier (**sndsig<f,b>**) command is similar to the **sndsig** command. However, the **sndsig*<f,b>*** command provides local ordering with respect to other commands within the same tag group (that is, commands with the same tag ID).

sndsigf ***CL, TG, TS, LSA, [EAH,] EAL***
sndsigb ***CL, TG, TS, LSA, [EAH,] EAL***

Implementation Note:

The **sndsig*<f,b>*** command can be implemented as a **put*<f,b>*** command.

8. Problem-State Memory-Mapped Registers

The problem-state (PS) memory-mapped registers define the set of facilities that an application running in problem state can access. An application is in problem state when MSR[PR] = '1' for a PowerPC Processor Element (PPE) or when MFC_SR1[PR] = '1' for a Synergistic Processor Element (SPE). Each SPE has a complete set of the registers as described in this section. The access privileges for each SPE are independent because the registers are allocated in different real address pages. Access to these registers is controlled by privileged software when mapping the corresponding address into the effective address of an application in the page table.

If an SPE is used as a privileged resource (that is, when MFC_SR1[PR] = '0'), access to these registers should also be marked as privileged. For more information about access privileges, see *PowerPC Architecture, Book III*.

The Cell Broadband Engine Architecture (CBEA) is flexible enough that, with proper settings in the page table, the problem-state memory-mapped registers listed in *Table 8-1* can be managed by PPEs, by other processors, or by other SPEs.

In the leftmost column of *Table 8-1*, the offset is relative to a starting location. That starting location is calculated using an implementation-dependent base address register (BP_Base), an SPE number, the problem-state area for an SPE, and so forth, as defined in *Table A-1 CBEA-Compliant Processor Memory Map* on page 294. See *Table A-1* for the mapping of all the registers defined by the Cell Broadband Engine Architecture in the real address space. For the reader's convenience, *Table A-2 SPE Problem State Memory Map* on page 296 duplicates *Table 8-1* below.

Table 8-1. SPE Problem-State Memory Map (Page 1 of 2)

Offset (Hexadecimal)	Register	Description	Access Type
Multisource Synchronization Area			
x'00000'	MFC_MSSync	MFC Multisource Synchronization Register (see page 109)	Read/Write
Memory Flow Controller (MFC) Proxy Command Parameter Area			
x'03000'	Reserved	Reserved for future expansion	Reserved
x'03004'	MFC_LSA	MFC Local Storage Address Register (see page 85)[1]	Write Only
x'03008'	MFC_EAH	MFC Effective Address High Register (see page 86)[1]	Write Only
x'0300C'	MFC_EAL	MFC Effective Address Low Register (see page 87)[1]	Write Only
x'03010'	MFC_Size	MFC Transfer Size Register (see page 84)[1, 2] (Upper 16 bits of register).	Write Only
	MFC_Tag	MFC Command Tag Register (see page 83)[1, 2] (Lower 16 bits of register)	Write Only
x'03014'	MFC_ClassID	MFC Class ID Register (see page 82)[1, 3] (Upper 16 bits of register for write)	Write Only
	MFC_Cmd	MFC Command Opcode Register (see page 81)[1, 2] (Lower 16 bits of register for write)	Write Only
	MFC_CMDStatus	MFC Command Status Register (see page 90)(all 32 bits for read)	Read Only

1. Reading of these registers should be allowed for diagnostic purposes.
2. Both the MFC_Size and MFC_TAG registers must be written with a single 32-bit store instruction.
3. Both the MFC_ClassID and MFC_Cmd registers must be written with a single 32-bit store instruction.

Table 8-1. SPE Problem-State Memory Map (Page 2 of 2)

Offset (Hexadecimal)	Register	Description	Access Type
MFC Proxy Status and Command Queue Control Area			
x'03020':x'030FF'	Reserved	Reserved	
x'03104'	MFC_QStatus	MFC Queue Status Register (see page 91)	Read Only
x'03204'	Prxy_QueryType	Proxy Tag-Group Query Type Register (see page 93)	Read/Write
x'0321C'	Prxy_QueryMask	Proxy Tag-Group Query Mask Register (see page 94)	Read/Write
x'0322C'	Prxy_TagStatus	Proxy Tag-Group Status Register (see page 95)	Read Only
x'03330':x'03FFF'	Reserved	Reserved	
Synergistic Processor Unit (SPU) Control Area			
x'04004'	SPU_Out_Mbox	SPU Outbound Mailbox Register (see page 102)	Read Only
x'0400C'	SPU_In_Mbox	SPU Inbound Mailbox Register (see page 103)[1]	Write Only
x'04014'	SPU_Mbox_Stat	SPU Mailbox Status Register (see page 104)	Read Only
x'0401C'	SPU_RunCntl	SPU Run Control Register (see page 96)	Read/Write
x'04024'	SPU_Status	SPU Status Register (see page 97)	Read Only
x'04034'	SPU_NPC	SPU Next Program Counter Register (see page 99)	Read/Write
x'04038':x'13FFF'	Reserved	Reserved	
Signal-Notification Area			
x'1400C'	SPU_Sig_Notify_1	SPU Signal Notification 1 Register (see page 106)	Read/Write
x'14010':x'1BFFF'	Reserved	Reserved	
x'1C00C'	SPU_Sig_Notify_2	SPU Signal Notification 2 Register (see page 107)	Read/Write
x'1C010':x'1FFFF'	Reserved	Reserved	

1. Reading of these registers should be allowed for diagnostic purposes.
2. Both the MFC_Size and MFC_TAG registers must be written with a single 32-bit store instruction.
3. Both the MFC_ClassID and MFC_Cmd registers must be written with a single 32-bit store instruction.

8.1 MFC Proxy Command Parameter Registers

The following subsections describe the MFC proxy command parameter registers:

- *Section 8.1.1 MFC Command Opcode Register* beginning on page 81
- *Section 8.1.2 MFC Class ID Register* beginning on page 82
- *Section 8.1.3 MFC Command Tag Register* beginning on page 83
- *Section 8.1.4 MFC Transfer Size Register* beginning on page 84
- *Section 8.1.5 MFC Local Storage Address Register* beginning on page 85
- *Section 8.1.7 MFC Effective Address Low Register* beginning on page 87
- *Section 8.1.6 MFC Effective Address High Register* beginning on page 86

8.1.1 MFC Command Opcode Register (MFC_Cmd)

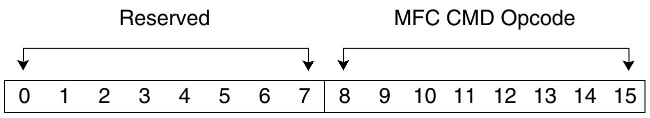
The MFC Command Opcode Register determines the operation to be performed. The validity of the opcode is checked asynchronously to the instruction stream. If the MFC proxy command or any of its parameters are invalid, MFC proxy command queue processing is suspended. If the interrupt is enabled, an invalid MFC command interrupt is sent. For more information about interrupts, see *Section 21 Interrupt Facilities* beginning on page 261.

Software must avoid programming practices that enqueue commands with forward dependencies on commands that are enqueued later. Software with this type of dependency can create a deadlock, because of the number of available slots in the MFC proxy command queues. In addition, while queue depth is implementation dependent, software must not be written to require a specific queue depth.

Software can determine the number of queue slots available in the MFC proxy command queue by reading the MFC Queue Status Register (see page 91). The value returned is the number of available queue slots. Software can use this value to avoid repeating the queuing sequence for a full MFC proxy command queue.

Section 8.2 MFC Proxy Command Issue Sequence on page 88 describes the queuing sequence for MFC proxy commands.

Access Type	MMIO: Read[1]/Write
Base Address Offset	(BP_Base I PS(n)) + x‘03014’; where n is an SPE number (lower 16 bits).



Bits	Field Name	Description
0:7	Reserved	Should be set to zeros. If bit 0 is set to ‘1’, the opcode is reserved.
8:15	MFC CMD Opcode	MFC proxy command opcode.

Note: The MFC Command Opcode Register must be written to the MFC proxy command queue along with the MFC Class ID Register using a single 32-bit store instruction. The MFC Command Opcode Register is the lower 16 bits of the 32-bit word. The upper 8 bits of the MFC Command Opcode field are reserved. If the MSb (that is, bit 0 of this field) is set to ‘1’, the proxy command is reserved and can be used for an implementation-dependent function.

Programming Note:

The total number of queue slots is implementation dependent and varies between implementations. For portability of an application, the queuing sequence for MFC proxy commands and the method to determine the number of queue slots available should be provided as a macro or as a library function that uses a configuration-dependent method for queuing commands.

1. When read, this register returns the value in the MFC Command Status Register (see page 90).

8.1.2 MFC Class ID Register (MFC_ClassID)

The MFC Class ID Register is used to specify the replacement class ID and the transfer class ID for each MFC proxy command. Software can use these IDs to improve the overall performance of the system.

The transfer class ID (TclassID) lets an implementation optimize MFC proxy command transfers based on the characteristics of the storage location. The replacement class ID (RClassID) lets an implementation influence cache replacement. For more information on the replacement class ID, see *Section 19 Cache Replacement Management Facility* on page 255. The exact function and mapping of the replacement class ID and the transfer class ID are implementation dependent. For more information on setup and use of these IDs, see the specific implementation documentation.

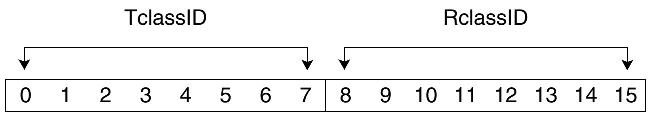
The MFC class ID performs the same function for commands that are issued to the MFC SPU command queue or to the MFC proxy command queue. Thus, its function is queue independent. For example, the RclassID affects which set in the translation lookaside buffer (TLB) is available to be replaced. The RClassID is a pointer into the replacement management table (RMT), which is shared between queues. Since the RMT is shared and the TLB is shared, the class ID is independent of the queue to which the command was issued. (A parameter such as the Tag parameter, on the other hand, is queue specific).

The contents of the MFC Class ID Register are not persistent and must be written for each MFC proxy command queuing sequence.

The validity of the MFC Class ID Register is not verified. The number of class IDs supported is implementation dependent. The default class ID (x‘00’) is used for all undefined or invalid class IDs. An invalid class ID parameter does not generate an interrupt.

Access Type MMIO: Read[1]/Write

Base Address Offset (BP_Base | PS(n)) + x‘03014’; where n is an SPE number (upper 16 bits).



Bits	Field Name	Description
0:7	TclassID	Transfer class identifier.
8:15	RclassID	Replacement class identifier.

Note: The MFC Class ID Register must be written along with the MFC Command Opcode Register (see page 81) using a single 32-bit store instruction. The MFC proxy command parameter is the lower 16 bits of a 32-bit word. The MFC class ID parameter is the upper 16 bits of the 32-bit word.

1. When read, this register returns the value in the MFC Command Status Register (see page 90).

8.1.3 MFC Command Tag Register (MFC_Tag)

The MFC Command Tag Register is used to specify an identifier for each proxy command or group of proxy commands. The tag can be any value between x‘0’ and x‘1F’. Tags have a purely local scope in the hardware.

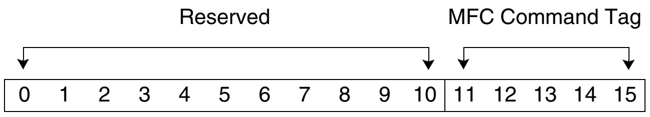
Any number of MFC proxy commands can be tagged with the same identification. MFC proxy commands tagged with the same ID are called a tag group. Tags are associated with commands written to a specific queue. Tags supplied to the MFC proxy command queue are independent of tags supplied to the MFC SPU command queue.

The contents of the MFC Command Tag Register are not persistent and must be written for each MFC proxy command queuing sequence.

The validity of this register is checked asynchronously to the instruction stream. If the upper bits (bits 0 through 10) are not set to zeros, the proxy command queue processing is suspended. If the interrupt is enabled, an invalid MFC proxy command interrupt is sent. For more information, see *Section 21 Interrupt Facilities* beginning on page 261.

Note: Software must write the MFC Command Tag Register for each command, even if the command does not support this parameter. Even though it is written, the tag is not used if the command does not support this parameter. For future compatibility, software should set the MFC proxy command tag to zero for all commands that do not support the MFC proxy command tag.

Access Type	MMIO: Write Only[1]
Base Address Offset	(BP_Base I PS(n)) + x‘03010’; where n is an SPE number (lower 16-bit).



Bits	Field Name	Description
0:10	Reserved	Set to zeros.
11:15	MFC Command Tag	MFC proxy command tag.

Note: The MFC Command Tag Register must be written along with the MFC Transfer Size Register (see page 84) using a single 32-bit store instruction. The MFC Command Tag parameter is the lower 16 bits of the 32-bit value written to the base address offset.

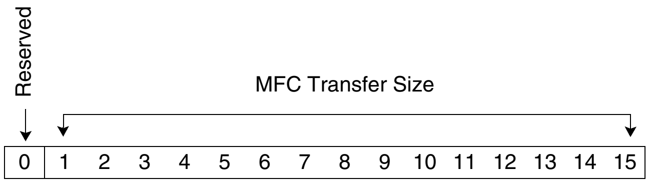
1. An implementation should support read access to this facility for diagnostic purposes.

8.1.4 MFC Transfer Size Register (MFC_Size)

The MFC Transfer Size Register is used to specify the size of an MFC transfer. The transfer size value can be 0, 1, 2, 4, 8, 16, or a multiple of 16 bytes to a maximum of 16 KB. The contents of the MFC Transfer Size Register are not persistent and must be written for each MFC proxy command enqueue sequence.

The validity of this register is checked asynchronously to the instruction stream. If the size is invalid, the MFC proxy command queue processing is suspended. If the interrupt is enabled, an MFC direct memory access (DMA) alignment interrupt is sent. For more information about interrupts, see *Section 21 Interrupt Facilities* beginning on page 261.

Access Type	MMIO: Write Only[1]
Base Address Offset	(BP_Base	PS(n)) + x‘03010’; where n is an SPE number (upper 16 bits).



Bits	Field Name	Description
0	Reserved	Set to zeros.
1:15	MFC Transfer Size	MFC transfer size. Allowable MFC transfer sizes follow: • 0, 1, 2, 4, and 8 bytes (naturally aligned), where the source and destination address *must* have the same 4 least-significant bits (see *Table 7-6 Command Errors and Alignment Errors* on page 61 for more information). • 16 bytes and multiples of 16 bytes up to 16 KB, where the source and destination addresses must be 16-byte (quadword) aligned.

Note: The MFC Transfer Size Register must be written along with the MFC Command Tag Register using a single 32-bit store instruction. The MFC Transfer Size Register is the upper 16 bits of the 32-bit value written to the base address offset.

Programming Note:

Typically, systems are more efficient when they perform transfers of a cache line or multiple cache lines. Excessive use of small transfers (less than a cache line) can result in poor bus and memory bandwidth utilization. In addition, when transferring multiple quadwords, better performance can typically be achieved when the effective address and local storage address are both aligned within the cache line. For example, if the cache line size is 128-bytes, optimal performance can be achieved when bits 25 through 27 of the 32-bit local storage address are equal to bits 57 thorough 59 of the 64-bit effective address. The lower 4 bits of the effective address and local storage address must be zero for quadword transfers.

The method that an application uses to determine the cache line size of the processor is implementation dependent. For more information, see the specific implementation documentation and software interface specifications.

1. An implementation should support read access to this facility for diagnostic purposes.

8.1.5 MFC Local Storage Address Register (MFC_LSA)

The MFC Local Storage Address Register is used to supply the SPU local storage address associated with an MFC proxy command to be queued. This address is used as the source or destination of the MFC transfer as defined in the MFC proxy command. For more information, see *Section 7 MFC Commands* beginning on page 55.

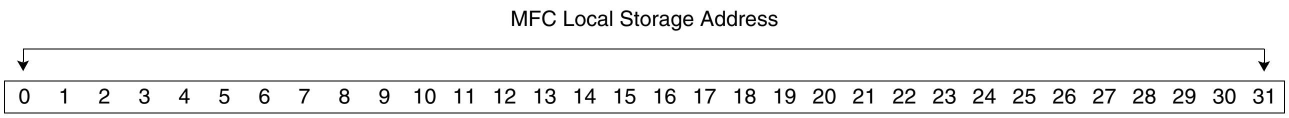
The contents of the MFC Local Storage Address Register are not persistent and must be written for each MFC proxy command queuing sequence.

The validity of this register is checked asynchronously to the instruction stream.

To be considered aligned, the 4 least-significant bits of the local storage address must match the least-significant 4 bits of the effective address. If the address is unaligned, the processing for both command queues is suspended. If the interrupt is enabled, an MFC DMA alignment interrupt is sent. For more information about interrupts, see *Section 21 Interrupt Facilities* beginning on page 261.

Access Type MMIO: Write Only[1]

Base Address Offset (BP_Base | PS(n)) + x‘03004’; where n is an SPE number.



Bits	Field Name	Description
0:31	MFC Local Storage Address	MFC local storage address. **Note:** The 4 least-significant bits of the local storage address must match the least-significant 4 bits of the effective address.

Programming Note:

Typically, systems are more efficient when they perform transfers of a cache line or multiple cache lines. Excessive use of small transfers (less than a cache line) can result in poor bus and memory bandwidth utilization. In addition, when transferring multiple quadwords, better performance can typically be achieved when the effective address and local storage address are both aligned within the cache line. For example, if the cache line size is 128-bytes, optimal performance can be achieved when bits 25 through 27 of the 32-bit local storage address are equal to bits 57 thorough 59 of the 64-bit effective address. The lower 4 bits of the effective address and local storage address must be zero for quadword transfers.

The method that an application uses to determine the cache line size of the processor is implementation dependent. For more information, see the specific implementation documentation and software interface specifications.

1. An implementation should support read access to this facility for diagnostic purposes.

8.1.6 MFC Effective Address High Register (MFC_EAH)

The MFC Effective Address High Register is used to specify the upper 32 bits of the 64-bit effective address for the MFC proxy command.[1] If translation is enabled in MFC State Register One (that is, MFC_SR1[R] is set to ‘1’), effective addresses are translated into real addresses by the memory management unit (MMU) address translation mechanism (for more information, see *PowerPC Architecture, Book III)*. If translation is disabled (the effective address equals the real address), the number of lower effective address bits that are valid is implementation dependent. For more information, see the specific implementation documentation.

The contents of the MFC Effective Address High Register are not persistent and must be written for each MFC proxy command enqueue sequence.

This parameter is optional. If the effective address high (EAH) is not written, then the hardware sets the EAH parameter to zero (that is, the address is between 0 and 2^{32}-1).

The validity of this parameter is checked asynchronously to the instruction stream. If the address is invalid (for example, due to a segment fault, a mapping fault, or a protection violation), processing of the MFC proxy command is suspended. An interrupt, if enabled, is sent. Processing of other commands in the queue continues if possible, subject to any fence or barrier commands.

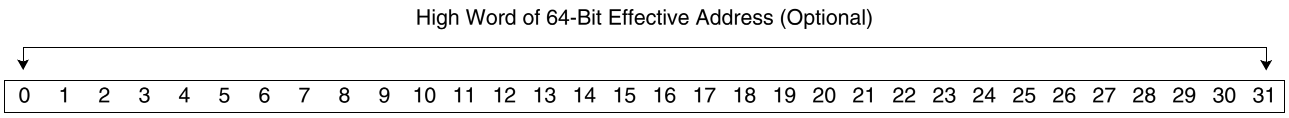
The following types of interrupts can be sent:

- If a segment fault occurs, an MFC data-segment interrupt is sent.
- If a mapping fault or a page protection violation occurs, an MFC data-storage interrupt is sent.

For more information about interrupts, see *Section 21 Interrupt Facilities* beginning on page 261.

Note: The validity of the effective address can be checked during transfers. Partial transfers can be performed before an invalid address is encountered and the exception is generated.

Access Type	MMIO: Write Only[2]
Base Address Offset	(BP_Base I PS(n)) + x‘03008’; where n is an SPE number.



Bits	Field Name	Description
0:31	High Word of 64-Bit Effective Address (Optional)	High word of the 64-bit effective address. • EAH is optional. If not written, zeros are used for the upper 32 bits of the 64-bit effective address. • If translation is disabled, the number of lower effective address bits that are valid is implementation specific. The remaining upper bits are required to be zero.

1. The effective address can be written using either one 64-bit store or two 32-bit stores. If written using 32-bit stores, the EAH must be written first. In general, 64-bit access to an address range that includes a 32-bit MMIO register is not allowed except when specified explicitly as in this instance.
2. An implementation should support read access to this facility for diagnostic purposes.

8.1.7 MFC Effective Address Low Register (MFC_EAL)

The MFC Effective Address Low Register is used to specify the lower 32 bits of the 64-bit effective address for the MFC proxy command.[1] If translation is enabled in the MFC State Register One (that is, MFC_SR1[R] is set to ‘1’), effective addresses are translated into real addresses by the MMU address translation mechanism (for more information, see *PowerPC Architecture, Book III*). If translation is disabled, the 64-bit address formed by MFC_EAH || MFC_EAL must be a supported address within the real address space of main storage. Handling of unsupported addresses is implementation dependent. For more information about the real address space of an implementation, see the specific implementation documentation.

The contents of the MFC Effective Address Low Register are not persistent and must be written for each MFC proxy command enqueue sequence.

For transfer sizes less than 16 bytes, MFC_EAL bits 28 through 31 must provide natural alignment based on the transfer size. For transfer sizes of 16 bytes or greater, bits 28 through 31 must be zeros. In addition to these limitations, bits 28 through 31 must match bits 28 through 31 of the MFC_LSA. If any of these conditions are not met, the MFC_EAL parameter is considered unaligned and is invalid.

The validity of this parameter is checked asynchronously to the instruction stream. If the address is invalid (for example, due to a segment fault, a mapping fault, a protection violation, or an alignment error), processing of the MFC proxy command is suspended. An interrupt, if enabled, is sent. Processing of other commands in the queue continues if possible, subject to any fence or barrier commands.

The following types of interrupts can be sent:

- If a segment fault occurs, an MFC data segment interrupt is sent.
- If a mapping fault or a page protection violation occurs, an MFC data-storage interrupt is sent.
- If the address is not aligned, a DMA alignment interrupt is sent.

For more information about interrupts, see *Section 21 Interrupt Facilities* beginning on page 261.

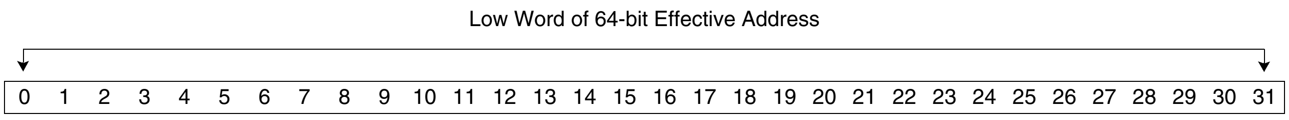
Notes:

- The validity of the effective address can be checked during transfers. Partial transfers can be performed before an invalid address is encountered and the exception is generated.
- For optimal performance of transfers of 128 bytes or more, the source and destination transfer address should be 128-byte aligned (bits 25 through 31 set to 0).

1. The effective address can be written using either one 64-bit store or two 32-bit stores. If written using 32-bit stores, the EAH must be written first. In general, 64-bit access to an address range that includes a 32-bit MMIO register is not allowed except when specified explicitly as in this instance.

Access Type MMIO: Write Only[1]

Base Address Offset (BP_Base | PS(n)) + x'0300C'; where n is an SPE number.



Bits	Field Name	Description
0:31	Low Word of 64-bit Effective Address	Low word of the 64-bit effective address. For transfer sizes less than 16 bytes, address bits 28 through 31 must provide natural alignment based on the transfer size. For transfer sizes of 16 bytes or greater, bits 28 through 31 must be zeros. If translation is disabled (the effective address equals the real address), some higher-order bits must also be zero, depending on the amount of real memory in the system.

Programming Note:

Typically, systems are more efficient when they perform transfers of a cache line or multiple cache lines. Excessive use of small transfers (less than a cache line) can result in poor bus and memory bandwidth utilization. In addition, when transferring multiple quadwords, better performance can typically be achieved when the effective address and local storage address are both aligned within the cache line. For example, if the cache line size is 128-bytes, optimal performance can be achieved when bits 25 through 27 of the 32-bit local storage address are equal to bits 57 thorough 59 of the 64-bit effective address. The lower 4 bits of the effective address and local storage address must be zero for quadword transfers.

The method that an application uses to determine the cache line size of the processor is implementation dependent. For more information, see the specific implementation documentation and software interface specifications.

8.2 MFC Proxy Command Issue Sequence

The CBEA requires software to follow a particular sequence to enqueue an MFC proxy command. If the correct sequence is not followed, an MFC proxy command is not queued and an invalid command sequence is posted in the MFC Command Status Register (see page 90) (that is, MFC_CMDStatus[RC] = '01' or '11').

The MFC Command Parameter Registers must be written in increasing address order. To enqueue an MFC proxy command from a PPE, the MFC parameters must first be written to the MFC proxy command address space in the following sequence:

1. Set the local storage address.
2. Set the MFC effective address.[2]

1. An implementation should support read access to this facility for diagnostic purposes.
2. The effective address can be written using either one 64-bit store or two 32-bit stores. If written using 32-bit stores, the EAH must be written first. In general, 64-bit access to an address range that includes a 32-bit MMIO register is not allowed except when specified explicitly as in this instance.

3. Set the MFC size and the MFC tag.[1]
4. Set the transfer class and replacement management class IDs and the MFC proxy command.[1]
5. Read the MFC proxy command status.[2]
   Reading the MFC proxy command status causes an attempt to enqueue the specified command and its parameters.
6. Restart the MFC proxy command issue sequence, starting at step 1, if the status indicates failure or if there is no slot available in the MFC proxy command queue.

These parameters are held in the corresponding registers. The read of the MFC proxy command status causes the data held in these registers to be enqueued, if the sequence has not been interrupted and if there is a slot in the MFC proxy command queue.

Implementation Note:

When the local storage address is set in step 1, hardware should set an internal state bit, CMD_Pending. This bit indicates that a command is pending for the queue. The CMD_Pending bit is reset if the required sequence for enqueuing an MFC proxy command is not followed.[3] An MFC proxy command fails if the CMD_Pending bit is reset when the MFC Command Status Register is read in step 5.[4] If the CMD_Pending bit is still set when the MFC Command Status Register is read in step 5, the command is enqueued. Once the command is enqueued, the CMD_Pending bit is reset.

8.3 MFC Proxy Command Queue Status and Control Registers

The MFC provides an MFC SPU command queue, as well as a separate MFC proxy command queue for PPEs and other devices. The commands and procedures for using the MFC SPU command queue are defined in *Section 9 Synergistic Processor Unit Channels* beginning on page 113. The registers defined in this section are the status and control registers for the MFC proxy command queue.

The MFC Proxy Command Parameter Registers (see page 80) along with the MFC Command Status Register (see page 90) are used for queuing an MFC proxy command to the MFC proxy command queue.

The procedure for queuing a proxy command is outlined in *Section 8.2 MFC Proxy Command Issue Sequence* beginning on page 88. These registers are intended to be used by software running on PPEs. However, they can be made available to other SPUs or to other devices in the system.

In addition to the MFC Command Status Register, the MFC provides registers for determining the number of proxy command slots available in the MFC proxy command queue. The MFC also provides registers for determining when a tag group in the MFC proxy command queue is complete.

1. The MFC Transfer Size Register (see page 84) and the MFC Command Tag Register (see page 83) are a pair of registers. The MFC Class ID Register (see page 82) and the MFC Command Opcode Register (see page 81) are a pair of registers. Each pair must be updated using a single store instruction.
2. Reading the MFC Command Status Register (see page 90) causes the parameters to be enqueued if the sequence is correct.
3. If the local storage address is set again, then CMD_Pending is set again instead of being reset.
4. An MFC proxy command can also fail if there is insufficient room in the MFC proxy command queue.

8.3.1 MFC Command Status Register (MFC_CMDStatus)

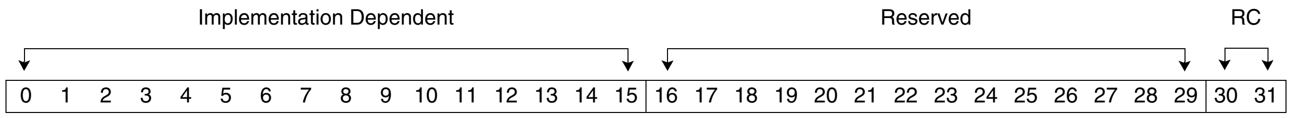
The MFC Command Status Register contains the return code from the last attempt to enqueue an MFC proxy command to the MFC proxy command queue.

The MFC Command Opcode Register (see page 81) and MFC Command Status Register are mapped to the same address. Ordering of the memory-mapped I/O (MMIO) accesses on these registers is maintained; therefore, no intervening **eieio** instruction or **mfceieio** command is required.

Note: The MFC Command Status Register is a 32-bit register (the upper 16 bits are implementation dependent). The MFC proxy command return code in the least-significant bits returns the proxy command status when read.

Access Type Read

Base Address Offset (BP_Base | PS(n)) + x'03014'; where n is an SPE number.



Bits	Field Name	Description
0:15	Implementation Dependent	Implementation dependent.
16:29	Reserved	Set to zeros.
30:31	RC	MFC proxy command return code. 00 Command enqueue successful. 01 Command enqueue failed due to sequencing error. 10 Command enqueue failed due to insufficient space in the MFC proxy command queue. The free space in the MFC proxy command queue is zero. 11 Command enqueue failed due to sequencing error. The free space in the MFC proxy command queue is zero.

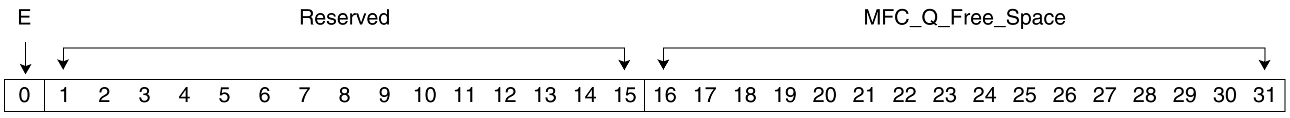
8.3.2 MFC Queue Status Register (MFC_QStatus)

The MFC Queue Status Register contains the current status of the MFC proxy command queue.
The E bit (‘0’) of this register indicates either that the MFC proxy command queue is empty, or that it contains valid proxy commands that have not finished processing. The lower 16 bits of this register return the number of entries available in the MFC proxy command queue. Zeros in these bits indicate that the queue is full.

The problem-state registers are mapped as both caching-inhibited and guarded. Thus, an enforce in-order execution of I/O (**eieio**) instruction is required between the store of a command and the load of its return code. Software must issue an **eieio** instruction before reading this register to ensure that the effects of all previously issued MMIO instructions or commands are reported correctly.

Access Type Read

Base Address Offset (BP_Base | PS(n)) + x‘03104’; where n is an SPE number.



Bits	Field Name	Description
0	E	MFC proxy command queue empty. All MFC operations are complete. 0 MFC proxy command queue contains commands. 1 MFC proxy command queue does not contain any commands.
1:15	Reserved	Reserved.
16:31	MFC_Q_Free_Space	MFC proxy command queue free space. This field contains the number of queue entries available. Software can use this field to set a loop count for the number of MFC proxy commands to enqueue. Software must not assume that a proxy command is enqueued based on the free space. Other conditions can cause the proxy command issue sequence to fail. For proper operation, software must follow the procedure outlined in *Section 8.2 MFC Proxy Command Issue Sequence* on page 88.

8.4 Proxy Tag-Group Completion Facility

Each MFC proxy command is tagged with a 5-bit identifier (that is, the MFC proxy command tag). The same identifier can be used for multiple MFC proxy commands. A set of commands in the same queue with the same identifier is called a tag group. Software can use this identifier to determine when a command or group of commands has completed. (That is, it can use the MFC proxy command tag to check, or to wait on, the completion of all queued commands for each tag group.) In addition, an interrupt can be sent to a processor or device upon the completion of one or more tag groups if the interrupt is enabled by privileged software.

When the status returned by a channel read instruction that targets the MFC Read Tag-Group Status Channel (see page 133) indicates that a **put** command is complete, the local storage accesses are complete. The accesses are ordered with respect to the SPU. However, the main storage accesses might not be complete. The accesses are not ordered with respect to other processors and devices. For a **get** command, both the local storage and main storage accesses are complete and ordered with respect to other processors and devices.

The following procedure polls for the completion of one or more tag groups in the MFC proxy command queue:

1. Issue the MFC proxy commands to the MFC proxy command queue.
2. Set the Proxy Tag-Group Query Mask Register to the groups of interest.
3. Read the Proxy Tag-Group Status Register.
4. If the value is nonzero, one of the tag groups of interest has completed. If polling for all the tag groups of interest is complete, perform an XOR between the proxy tag-group status value and the proxy tag-group query mask. A zero indicates that all groups of interest are complete.
5. Repeat steps 3 and 4 until the tag groups of interest are complete.

The following procedure generates an interrupt when one or more tag groups complete, if the interrupt is enabled by privileged software. The same procedure can be used to poll for the specific condition if the tag-group completion interrupt is disabled.

1. Perform steps 1 and 2 as defined above in the polling procedure.
2. Request a tag-group query by writing a value of '01' or '10' to the Proxy Tag-Group Query Type Register.
3. Wait for an interrupt to occur if the interrupt is enabled.

OR

4. Read the Proxy Tag-Group Query Type Register until a value of 0 is returned.

Notes:

1. The query type should not be changed between '01' and '10’ and the query mask should not be altered until the completion condition has been met. Doing so can produce an unexpected result. The query type can be set to zero to cancel an outstanding query request.
2. Privileged software must reset the interrupt status after the interrupt is received. A new interrupt is not generated until the interrupt status is reset and the MFC reenabled by writing the Proxy Tag-Group Query Type Register to a '01' or '10'.

8.4.1 Proxy Tag-Group Query Type Register (Prxy_QueryType)

Software uses the Proxy Tag-Group Query Type Register to request the MFC to detect a tag-group completion condition for the MFC proxy command queue. There are two possible conditions that can be requested: the completion of *any* enabled tag groups and the completion of *all* enabled tag groups. If the interrupt is enabled, a tag-group completion interrupt is sent to any processor or any device in the system. See *Section 21 Interrupt Facilities* beginning on page 261 for more information.

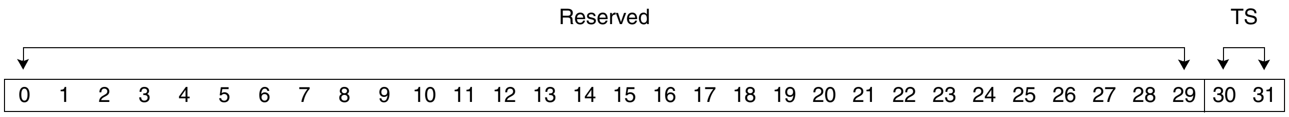
A read of this register returns the current status of the query. A nonzero value indicates that the completion condition has not occurred. When the completion condition is met, a read of this register returns zero, which indicates “query request complete.” If the interrupt is enabled by privileged software, a tag-group completion interrupt can be generated when the condition is met.

Writing zeros to this register while a request is still pending cancels the query request. Cancelling a query request also prevents a tag-group completion interrupt from occurring when the condition is met.

Software should not change the query type while a request is pending, except to cancel the request. Doing so can produce unexpected results.

Access Type Read/Write

Base Address Offset (BP_Base | PS(n)) + x‘03204’; where n is an SPE number.



Bits	Field Name	Description
0:29	Reserved	Reserved.
30:31	TS	Tag status update condition. 00 No query requested. 01 Set interrupt status upon completion of *any* enabled tag groups. 10 Set interrupt status only upon completion of *all* enabled tag groups. 11 Reserved.

8.4.2 Proxy Tag-Group Query Mask Register (Prxy_QueryMask)

The Proxy Tag-Group Query Mask Register selects the tag groups to be included in the query operation.

The data provided by this register is retained by the MFC until changed by a subsequent write to this register. Therefore, the data does not need to be respecified for each status query. If software modifies this mask when a query request is pending, the meaning of the results is ambiguous. A pending query request should be cancelled before this mask is modified. A query request can be cancelled by writing a value of ‘0’ (query complete) to the Proxy Tag-Group Query Type Register (see page 93).

Access Type Read/Write

Base Address Offset (BP_Base | PS(n)) + x‘0321C’; where n is an SPE number.

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 is not part of a query or status operation. 1 Tag group is part of a query or status operation.

8.4.3 Proxy Tag-Group Status Register (Prxy_TagStatus)

The Proxy Tag-Group Status Register contains the current status of the tag groups enabled in the Proxy Tag-Group Query Mask Register (see page 94). Only the status of the enabled tag groups is valid. The bit positions corresponding to disabled tag groups are set to '0'.

Software must issue an **eieio** instruction before reading this register to ensure that the effects of all previous stores are reflected in the read data.

Access Type	Read
Base Address Offset	(BP_Base I PS(n)) + x‘0322C’; where n is an SPE number.

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 has outstanding operations or is not part of query (that is, it is masked). 1 Tag group is complete; no outstanding operations.

8.5 SPU Control and Status Facilities

The following SPU control and status registers are provided:

- SPU Run Control Register
- SPU Status Register (see page 97)
- SPU Next Program Counter Register (see page 99)

8.5.1 SPU Run Control Register (SPU_RunCntl)

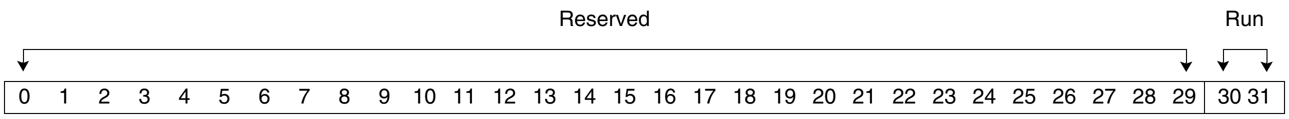
The SPU Run Control Register is used to start and stop the execution of instructions in the SPU. The SPU can dynamically change the state of the Run Status bit (that is, SPU_Status[R]).

The current status of the SPU run state is available in the SPU Status Register (see page 97).

Note: Performing a store operation to the SPU_RunCntl register is a context synchronizing operation in the SPU. See the *Synergistic Processor Unit Instruction Set Architecture* document for more information on synchronization.

Access Type Read/Write

Base Address Offset (BP_Base | PS(n)) + x‘0401C’; where n is an SPE number.

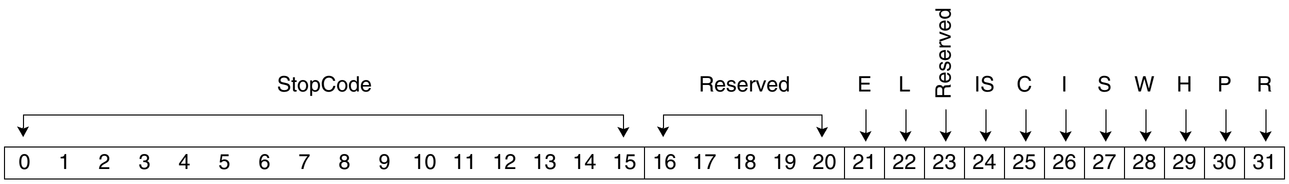


Bits	Field Name	Description
0:29	Reserved	Set to zeros.
30:31	Run	SPU run control. 00 SPU stop request (no instructions are issued). 01 SPU run request (instructions are issued if not stalled on condition). 10 SPU isolation exit request (an exit request is ignored if the SPU is not stopped or halted). 11 SPU isolation load request (a load request is ignored if the SPU is not stopped or halted, or if the load request enable is not set). **Notes:** • See the SPU Status Register for the current status. • If the Isolation state facility is not present, ‘10’ and ‘11’ are ignored.

8.5.2 SPU Status Register (SPU_Status)

The SPU Status Register is used to report the status (state) of an SPU. Software must issue an **eieio** instruction before reading this register to ensure that the effects of all previous stores are reflected in the read data.

Access Type	Read
Base Address Offset	(BP_Base I PS(n)) + x‘04024’; where n is an SPE number.



Bits	Field Name	Description
0:15	StopCode	SPU stop and signal type code. If the P bit (stop and signal indication) is set to a ‘1’, this field provides a copy of bits 18 through 31 of the SPU stop-and-signal instruction that resulted in the SPU stop. Bits 0 and 1 of this field always are zeros. If the P bit is not set, data in bits 0 through 15 is not valid. A stop-and-signal with a dependencies (STOPD) instruction, used for debugging, always sets this field to x'3FFF'.
16:20	Reserved	Reserved.
21	E	See *Section 11 SPU Isolation Facility* beginning on page 183. SPU Isolation Exit Status 0 SPU is not performing an isolation exit function. 1 SPU is performing an isolation exit function.
22	L	See *Section 11 SPU Isolation Facility* beginning on page 183. SPU Isolation Load Status 0 SPU is not performing an isolation load function. 1 SPU is performing an isolation load function.
23	Reserved	
24	IS	See *Section 11 SPU Isolation Facility* beginning on page 183. SPU Isolated State 0 SPU is not in an isolated state. 1 SPU is in an isolated state.
25	C	Invalid channel instruction detected. See note in *Section 9 Synergistic Processor Unit Channels* on page 113 for definition of an invalid channel instruction. 0 No invalid channel instruction detected. 1 Invalid channel instruction detected. SPU halted (stopped imprecisely). MFC notified of error condition.
26	I	Invalid instruction detected. 0 No invalid instruction detected. 1 Invalid instruction detected. SPU halted (stopped imprecisely). MFC notified of error condition.
27	S	SPU single-step status. See *Section 16.1 SPU Privileged Control Register* beginning on page 239. 0 SPU not stopped due to single-step mode. 1 SPU stopped after completion of an instruction in single-step mode.

Bits	Field Name	Description
28	W	SPU wait status. 0 SPU not waiting on a blocked channel. 1 SPU waiting on a blocked channel.
29	H	SPU halt instruction status. 0 SPU not halted (imprecise stopped) due to a halt instruction. 1 SPU halted (imprecise stopped) due to a halt instruction.
30	P	SPU program stop-and-signal status. (Applies to either STOP or STOPD instruction.) 0 SPU not stopped due to stop-and-signal instruction. 1 SPU stopped due to stop-and-signal instruction.
31	R	SPU run status. 0 SPU stopped or halted. 1 SPU running.

Programming Note:

A read of this register provides a current view of the SPU state. The status read can change dynamically, if the SPU is running. When the SPU is restarted bits, C, I, S, H, and P bits are cleared (that means that software does not have to clear these bits before restarting).

If the SPU is stopped, the status remains static until software changes it. If the SPU was stopped under PPE control while waiting on a blocked channel, the SPU wait status is set in conjunction with the SPU stopped status.

When a stop-and-signal instruction is executed by the SPU, the least-significant 14 bits of the instruction are copied to bits 2 through 15 of the SPU Status Register as a STOP type code. Bits 0 and 1 of the SPU Status Register are always zeros. It is recommended that a value of x‘0’ for the STOP type be reserved for “DATA Executed as Instruction.” It is further recommended that type values having the most-significant bit set (bit 2) be reserved for runtime or privileged services.

A type value of x‘3FFF’ should be reserved for debugger breakpoints, because that value is hard wired on a **STOPD** instruction.

Type codes with bit 2 of the type code set to zeros (x‘0001’ - x‘1FFF’) are available for application use. These definitions are not enforced but are recommended conventions that could eventually become part of the CBEA-compliant application binary interface (ABI).

8.5.3 SPU Next Program Counter Register (SPU_NPC)

The SPU Next Program Counter Register contains the address from which an SPU starts executing when the Run Control bit is set in the SPU Run Control Register (see page 96).

Access to this register is available in all states. Writes to this register update the contents only when the SPU is in the stopped and nonisolated state (SPU_Status[R] = '0', SPU_Status[IS] = '0']). Writes to this register have no effect if the SPU is an isolated state (SPU_Status[IS] = '1') or if the SPU is currently in the running state. When the SPU is in the nonisolated and stopped state, a read of this register returns the local storage address of the next instruction to be executed if the SPU is started without an intervening write to this register. In addition, the least-significant bit of the data returned indicates the starting SPU interrupt enable or disable state. When a read of this register is performed while the SPU is in an isolated state, a value of all zeros is always returned. The data returned for a read of this register when the SPU is in the running and nonisolated state (SPU_Status[R] = '1', SPU_Status[IS] = '0']) is undefined.

When the SPU has stopped execution, the hardware automatically updates the value in this register with the address of the next instruction to be executed and with the current SPU interrupt enabled or disabled state if execution is to be resumed with an SPU start command. The SPU can be stopped by any of the following means:

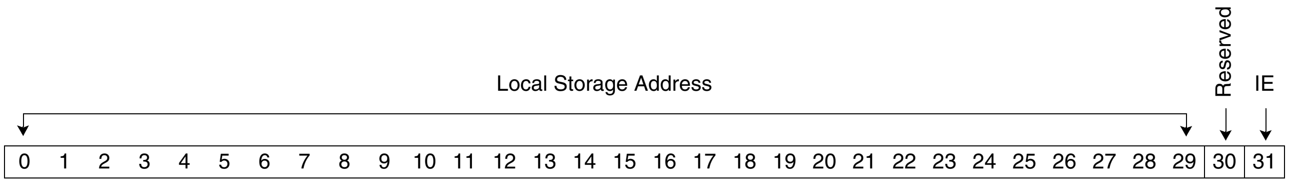
- The execution of SPU conditional halt instructions
- An SPU error
- The execution of an SPU stop-and-signal instruction
- The execution of a single instruction step
- Resetting the Run Control bit in the SPU Run-Control Register

If the stop was due to a stop-and-signal instruction in nonisolated state, the location of the actual stop-and-signal instruction can be determined by masking the enable or disable interrupt state bit (LSb), and then subtracting 4 from the value read from this register. For proper operation, software must ensure that the SPU program execution has stopped in a nonisolated state by reading and checking the values in the SPU Status Register before reading the SPU Next Program Counter Register.

To resume execution of a program, software writes the SPU Run Control Register to the appropriate value or issues an MFC proxy command with the S (start SPU) option. In a nonisolated state, to resume execution at a different point in the program or if a new program is loaded into the SPU, software should write this register to set the SPU program counter to the address in local storage of the next instruction to be executed. Software also writes this register to set the initial interrupt enable or disable state in effect when the SPU starts. It is not possible to affect the SPU execution address or state with this facility when the SPU is in an isolated state.

Access Type Read/Write

Base Address Offset (BP_Base | PS(n)) + x‘04034’; where n is an SPE number.



Bits	Field Name	Description
0:29	Local Storage Address	This field contains the local storage address of the first instruction issued when the Run Control bit in the SPU Run Control Register is enabled.
30	Reserved	Reserved
31	IE	Interrupt enable state. 0 SPU interrupts disabled at start. 1 SPU interrupts enabled at start.

8.6 Mailbox Facility

The MFC provides a set of mailbox queues between the SPU and other processors and devices. Each mailbox queue has an SPU channel assigned as well as a corresponding MMIO register. SPU software accesses the mailbox queues by using SPU channel instructions. Other processors and devices access the mailbox queues by using one of the MMIO registers. In addition to the queues, the MFC provides queue status, mailbox interrupts, and SPU event notification for the mailboxes. Collectively, the MMIO registers, channels, status, interrupts, mailbox queues, and events are called the mailbox facility.

The MFC provides two mailbox queues for sending information from the SPU to another processor or to other devices: the SPU outbound mailbox queue and the SPU outbound interrupt mailbox queue. These mailbox queues are intended for sending short messages to the PPEs (for example, return codes or status).

Data written by the SPU to one of these queues using a write channel (**wrch**) instruction is available to any processor or device by reading the corresponding MMIO register. A write channel (**wrch**) instruction that can target the SPU Write Outbound Interrupt Mailbox Channel (see page 140) can also cause an interrupt to be sent to a processor, or to another device in the system.

An MMIO read from either of these queues (SPU outbound mailbox or SPU outbound interrupt mailbox) can set an SPU event, which in turn causes an SPU interrupt. See *Section 9.11 SPU Event Facility* beginning on page 150 for a description of the SPU event facility. See *Section 9.12 SPU Event Definitions* beginning on page 163 for a description of the events that can be caused.

One mailbox queue is provided for either an external processor or other devices to send information to the SPU: the SPU inbound mailbox queue. This mailbox queue is intended to be written by a PPE. However, other processors, SPUs, or other devices can also use this mailbox queue. Data written by a processor or another device to this queue using an MMIO write is available to the SPU by reading the SPU Read Inbound Mailbox Channel (see page 141).

An MMIO write to the SPU Inbound Mailbox Register can set an SPU event, which in turn can cause an SPU interrupt.

The rest of this section describes the following registers:

- SPU Outbound Mailbox Register (see page 102)
- SPU Inbound Mailbox Register (see page 103)
- SPU Mailbox Status Register (see page 104)

For information about the Privilege 2 Mailbox Interrupt MMIO Register and the channels that correspond to these registers, see:

- *Section 15.12 SPU Outbound Interrupt Mailbox Register* beginning on page 237
- *Section 9.5.1 SPU Write Outbound Mailbox Channel* beginning on page 139
- *Section 9.5.2 SPU Write Outbound Interrupt Mailbox Channel* beginning on page 140
- *Section 9.5.3 SPU Read Inbound Mailbox Channel* beginning on page 141

8.6.1 SPU Outbound Mailbox Register (SPU_Out_Mbox)

The SPU Outbound Mailbox Register is used to read 32 bits of data from the corresponding SPU outbound mailbox queue. The SPU Outbound Mailbox Register has a corresponding SPU Write Outbound Mailbox Channel (see page 139) for writing data into the SPU outbound mailbox queue.

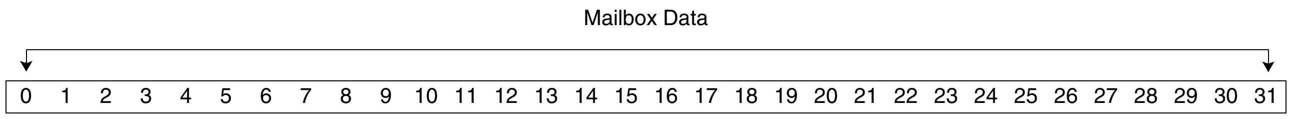
A write channel (**wrch**) instruction that targets the SPU outbound mailbox queue loads the 32 bits of data specified in the instruction into the SPU outbound mailbox queue for other processors or other devices to read. If the SPU outbound mailbox queue is full, the SPU stalls on the write channel (**wrch**) instruction that targets to this queue until an MMIO read from this mailbox register occurs. An MMIO read of this register always returns the information in the order it was written by the SPU. The information returned on a read from an empty SPU outbound mailbox queue is undefined.

The number of entries in the SPU outbound mailbox queue (or queue depth) is implementation dependent.

An MMIO read of the SPU Mailbox Status Register (see page 104) returns the status of the mailbox queues. The number of valid queue entries in the SPU outbound mailbox queue is given in the SPU_Out_Mbox_Count field of the SPU Mailbox Status Register (that is, SPU_Mbox_Stat [SPU_Out_Mbox_Count]).

An MMIO read of the SPU Outbound Mailbox Register sets a pending SPU outbound mailbox available event. If the amount of data remaining in the mailbox queue is below an implementation-dependent threshold and this condition is enabled (that is, SPU_WrEventMask[Le] is set to ‘1’), the SPU Read Event Status Channel is updated (that is, SPU_RdEventStat[Le] is set to ‘1’). Its channel count is set to 1, which causes an SPU outbound mailbox available event.

Access Type	MMIO: Read
Base Address Offset	(BP_Base | PS(n)) + x‘04004’; where n is an SPE number.



Bits	Field Name	Description
0:31	Mailbox Data	Application-specific mailbox data Each application can uniquely define the mailbox data.

Implementation Note:

The MFC must not acknowledge a write to the SPU Write Outbound Mailbox Channel (see page 139) until a PPE or other device has read the contents of the mailbox.

8.6.2 SPU Inbound Mailbox Register (SPU_In_Mbox)

The SPU Inbound Mailbox Register is used to write 32 bits of data into the corresponding SPU inbound mailbox queue. The SPU inbound mailbox queue has a corresponding SPU Read Inbound Mailbox Channel (see page 141) for reading data from the queue.

A read channel (**rdch**) instruction that targets the SPU Read Inbound Mailbox Channel loads the 32 bits of data from the SPU inbound mailbox queue into the SPU register specified by the read channel (**rdch**) instruction. The SPU cannot read from an empty mailbox. If the SPU inbound mailbox queue is empty, the SPU stalls on a read channel (**rdch**) instruction to this channel until data is written to the mailbox. A read channel (**rdch**) instruction to this channel always returns the information in the order it was written by PPEs or by other processors and devices.

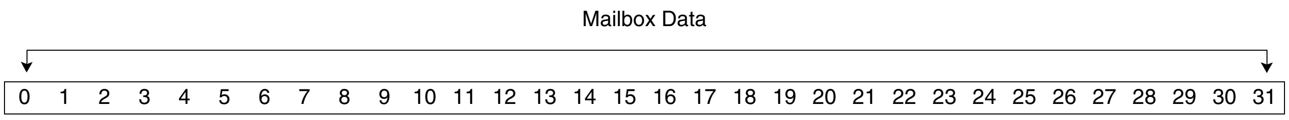
The number of entries in the SPU inbound mailbox queue (or queue depth) is implementation dependent.

An MMIO read of the SPU Mailbox Status Register (see page 104), returns the state of the mailbox queues. The number of available queue locations in the SPU inbound mailbox queue is given in the SPU_In_Mbox_Count field of the SPU Mailbox Status Register (that is, SPU_Mbox_Stat [SPU_In_Mbox_Count]). Software should check the SPU Mailbox Status Register before writing to the SPU_In_Mbox to avoid overrunning the SPU mailbox.

An MMIO write of the SPU Inbound Mailbox Register sets a pending SPU mailbox event. If enabled (that is, SPU_WrEventMask[Mbox] =‘1’), the SPU Read Event Status Channel is updated and its channel count is set to 1, which causes an SPU inbound mailbox available event.

Access Type MMIO: Write Only[1]

Base Address Offset (BP_Base | PS(n)) + x‘0400C’; where n is an SPE number.



Bits	Field Name	Description
0:31	Mailbox Data	Application-specific mailbox data Each application can uniquely define the mailbox data.

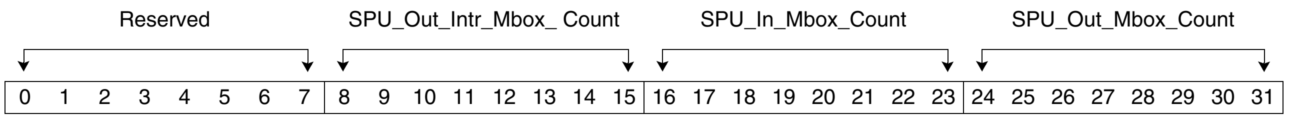
1. Read is only supported for diagnostic purposes.

8.6.3 SPU Mailbox Status Register (SPU_Mbox_Stat)

The SPU Mailbox Status Register contains the current state of the mailbox queues in the corresponding SPE. Reading this register has no effect on the state of the mailbox queues.

Access Type Read

Base Address Offset (BP_Base | PS(n)) + x‘04014’; where n is an SPE number.



Bits	Field Name	Description
0:7	Reserved	Set to zeros.
8:15	SPU_Out_Intr_Mbox_ Count	Number of valid entries in the SPU outbound interrupt mailbox queue (optional for a one-deep interrupt mailbox) The SPU_Out_Intr_Mbox_ Count value is incremented when the SPU writes the SPU Write Outbound Interrupt Mailbox Channel (see page 140). It is decremented when a processor or other device reads the SPU Outbound Interrupt Mailbox Register (see page 237). The number of entries supported is implementation dependent.
16:23	SPU_In_Mbox_Count	Number of available entries in the SPU inbound mailbox queue The SPU_In_Mbox_Count value is decremented when a processor or other device writes the SPU Inbound Mailbox Register. It is incremented when the SPU reads the SPU Read Inbound Mailbox Channel. The number of entries supported is implementation dependent.
24:31	SPU_Out_Mbox_Count	Number of valid entries in the SPU outbound mailbox queue The SPU_Out_Mbox_Count value is incremented when the SPU writes the SPU Write Outbound Mailbox Channel. It is decremented when a processor or other device reads the SPU Outbound Mailbox Register. The number of entries supported is implementation dependent.

8.7 SPU Signal Notification Facility

The SPU signal notification facility sends signals, such as a buffer completion flag, to an SPU from other processors and devices in the system. The CBEA provides two independent signal notification facilities: SPU Signal Notification 1 and SPU Signal Notification 2.

Each facility consists of one register and one channel:

- SPU Signal Notification 1 Register (see page 106) and SPU Signal Notification 1 Channel (see page 143)
- SPU Signal Notification 2 Register (see page 107) and SPU Signal Notification 2 Channel (see page 144)

Signals are issued by an SPU using a set of send signal commands (**sndsig*[<f,b>]***) with the effective address of the Signal Notification Register associated with the SPU to which the signal is sent. (For information about the commands, see *Section 7.9.4 Send Signal Command* on page 77 and *Section 7.9.5 Send Signal with Fence or with Barrier Command* on page 78.)

PPEs and other devices that do not support send signal commands simulate sending a signal notification by performing an MMIO write to the SPU Signal Notification Register associated with the SPU to which the signal is to be sent.

Each of the signal notification facilities can be programmed to either an overwrite mode, which is useful in a one-to-one signalling environment, or to a logical OR mode, which is useful in a many-to-one signalling environment. The mode for each channel is set in the SPU Configuration Register (see page 245). Performing either a send signal command or an MMIO write that targets a signalling register programmed to overwrite mode sets the contents of the associated channel to the data of the signalling operation. It also sets the corresponding channel count to 1. In logical OR mode, the data of the signalling operation is ORed with the current contents of the channel, and the corresponding channel count is set to 1.

In addition, the signal notification registers are used as the effective address pointer to the image loaded when requesting an isolation load. For more information, see *Section 11 SPU Isolation Facility* beginning on page 183. In these cases, SPU Signal Notification 1 Register contains the upper 32 bits of the 64-bit effective address. SPU Signal Notification 2 Register contains the least-significant 32 bits. For proper operation of an isolation load request, software must place the SPU signal notification facility in overwrite mode before it sets the effective address.

8.7.1 SPU Signal Notification 1 Register (SPU_Sig_Notify_1)

Access Type	MMIO: Read/Write
Base Address Offset	(BP_Base | PS(n)) + x‘1400C’; where n is an SPE number.

SigCntlWord

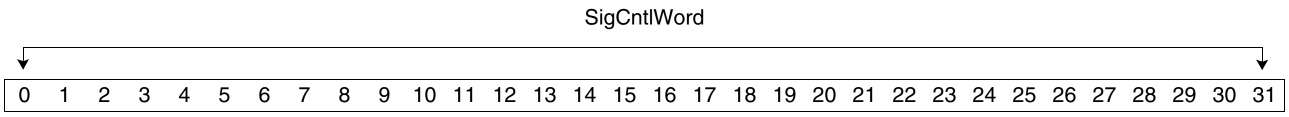
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

Bits	Field Name	Description
0:31	SigCntlWord	Signal-control word The application defines the data. It can either be ORed with the previous value, or it can overwrite the previous value. For more information, see *Section 16.4 SPU Configuration Register* beginning on page 245. The current contents are reset to zero when the SPU reads the value of the corresponding channel using an SPU read channel (**rdch**) instruction.

8.7.2 SPU Signal Notification 2 Register (SPU_Sig_Notify_2)

Access Type MMIO: Read/Write

Base Address Offset (BP_Base | PS(n)) + x‘1C00C’; where n is an SPE number.



Bits	Field Name	Description
0:31	SigCntlWord	Signal-control word The application defines the data. It can either be ORed with the previous value, or it can overwrite the previous value. For more information, see *Section 16.4 SPU Configuration Register* beginning on page 245. The current contents are reset to zero when the SPU reads the value of the corresponding channel using an SPU read channel (**rdch**) instruction.

8.8 MFC Multisource Synchronization Facility

The MFC multisource synchronization facility achieves cumulative ordering across the local storage and main storage address domains. The term cumulative ordering refers to the ordering of storage accesses that are performed by multiple sources (that is, two or more processors or units) with respect to another processor or unit. Standard PowerPC ordering rules apply to storage accesses that are performed by one processor or unit with respect to another processor or unit. See *PowerPC Architecture, Book II* for additional details.

Note: The MFC multisource synchronization facility is not an MFC command; it is a mechanism in an SPE.

Cumulative ordering is ensured when all accesses are performed in the main storage address domain and the proper synchronization instructions and commands are performed. Cumulative ordering cannot be ensured *just* by performing the proper synchronization instructions and commands when accesses are performed from *both* the main storage and the local storage address domains. When cumulative ordering is needed for accesses performed in both domains, software must use the MFC multisource synchronization facility and the proper synchronization instructions and commands to ensure all data is visible in both domains.

The MFC multisource synchronization facility consists of the following components:

- The MFC Multisource Synchronization Register (see page 109), which allows processors or devices to control synchronization from the main storage address domain
- The MFC Write Multisource Synchronization Request Channel (see page 149), which allows an SPU to control synchronization from the local storage address domain

The MFC multisource synchronization facility ensures that all transfers to or from the associated MFC that are received before the MFC multisource synchronization requests are completed. This facility does not ensure that read data is visible at the destinations when the associated MFC is the source. Synchronization requests caused by a write of the MFC Multisource Synchronization Register are independent of synchronization requests caused by a write of the MFC Write Multisource Synchronization Request Channel.

8.8.1 MFC Multisource Synchronization Register (MFC_MSSync)

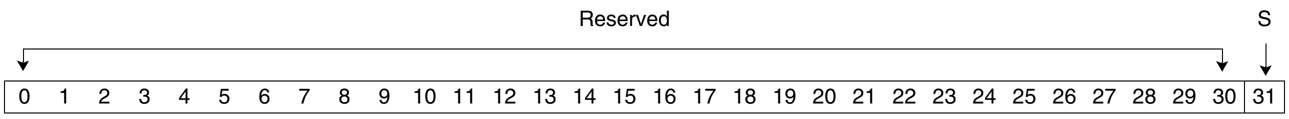
The MFC Multisource Synchronization Register is part of the MFC multisource synchronization facility, as is the MFC Write Multisource Synchronization Request Channel.

Writing any value to the MFC Multisource Synchronization Register requests synchronization. At the time of the write, the MFC starts to track all outstanding transfers to the corresponding SPE. When read, this register returns the current status of the last request. When all outstanding transfers to the corresponding SPE that were received before the last write of the MFC Multisource Synchronization Register are complete, a value of 0 is returned for an MMIO read of this register.

To use the MFC Multisource Synchronization Register, a program must perform the following steps:

1. Write to the MFC Multisource Synchronization Register.
2. Poll the MFC Multisource Synchronization Register until a value of 0 is received.

Access Type	MMIO: Read/Write
Base Address Offset	(BP_Base I PS(n)) + x‘00000’; where n is an SPE number.



Bits	Field Name	Description
0:30	Reserved	Set to zeros.
31	S	MFC Multisource Synchronization Status 0 All transfers received before MFC_MSSync register write are complete. 1 All transfers received before MFC_MSSync register write are not complete.

Programming Note:

Use of the MFC Multisource Synchronization Register is required for swapping context on an SPE. After stopping an SPU, privileged software must prevent any new transfers from being initiated to the SPE by unmapping the associated resources. Next, privileged software must use the MMIO MFC Multisource Synchronization Register to ensure the completion of all outstanding transfers. This has the side effect of ensuring the MFC Write Multisource Synchronization Request Channel count is 1 and the SPU Read Event Status is updated.

The following examples illustrate when the different multisource synchronization facilities are required. These examples show an I/O device storing data to the local storage alias area followed by a store to main storage. A second processor reads main storage and stores to another location in the local storage alias area. Cumulative ordering requires that the value loaded from location X by the SPU must be 1.

The first example illustrates the use of the MFC Multisource Synchronization Register. The second illustrates the use of the MFC Write Multisource Synchronization Request Channel. In both of these examples, locations X, Y, and Z all have an initial value of 0. A third example illustrates the use of the MFC multisource synchronization facility when starting an SPU.

Without the use of these facilities, the SPU might not receive a ‘1’ when reading the local storage location X.

Example 1. MFC Multisource Synchronization Register:

I/O Device (This example assumes that I/O transfers, including interrupts, are always processed in order.)
- Stores 1 to local storage alias X.
- Stores 2 to main storage location Y.

Processor
- Loops loading from main storage location Y until a value of 2 is obtained.
- Stores to the MFC Multisource Synchronization Register.
- Polls the MFC Multisource Synchronization Register until a zero is obtained.
- Stores 3 to local storage alias Z.

SPU
- Loops loading from local storage location Z until a value of 3 is obtained.
- Loads from local storage location X and obtains 1.

Example 2. MFC Write Multisource Synchronization Request Channel:

I/O Device (This example assumes that I/O transfers, including interrupts, are always processed in order.)
- Stores 1 to local storage alias X.
- Stores 2 to main storage location Y.

Processor
- Loops loading from main storage location Y until a value of 2 is obtained.
- Stores 3 to local storage alias Z.

SPU
- Loads from local storage location Z and obtains 3.
- Writes to the SPU_WrEventAck Channel with bit 19 set to acknowledge a previous multisource synchronization event.
- Writes to the SPU_WrEventMask Channel with bit 19 set to enable the multisource synchronization event.
- Channel writes to the MFC Write Multisource Synchronization Request Channel.
- Waits for a multisource synchronization event to occur. (The wait can be coded as a read of the count for MFC_WrMSSyncReq Channel, reading the SPU Read Event Status Channel (see page 153), or with a **bisled** instruction.)
- Loads from local storage location X and obtains 1.

Example 3. MFC Multisource Synchronization Facility:

I/O Device (This example assumes that I/O transfers, including interrupts, are always processed in order.)
- Stores 1 to local storage alias X.
- Interrupts the processor.

Processor
- After receiving the interrupt, performs any necessary instructions or procedures to guarantee that any stores from an I/O device are complete. (This is required if interrupts are not ordered with previous stores from the I/O device.)
- Stores to the MFC Multisource Synchronization Register.
- Polls the MFC Multisource Synchronization Register until a zero is obtained.
- Stores x‘01’ to the SPU Run Control Register to start the SPU.

SPU
- Starts to execute instructions, which load from local storage location X, and obtains 1.

Note: Frequent use of a single MFC multisource synchronization facility by two or more processors or devices can result in a livelock condition. The livelock occurs when a read from a processor or from a device never returns zero due to a synchronization request from other processors or devices.

Implementation Note:

The MFC Multisource Synchronization Register and the MFC Write Multisource Synchronization Request Channel must be treated independently. A synchronization request from the MFC proxy facility must not have any effects on the channel request and vice versa.

9. Synergistic Processor Unit Channels

In the Cell Broadband Engine Architecture (CBEA), channels are used as the primary interface between the synergistic processor unit (SPU) and the memory flow controller (MFC). The SPU channel access facility (see page 242) is used to configure, save, and restore the SPU channels. The SPU Instruction Set Architecture (ISA) provides a set of channel instructions for communication with external devices through a channel interface (or SPU channels). *Table 9-1* lists these instructions.

Table 9-1. SPU Channel Instructions

Channel Instruction	Instruction Mnemonic	Operational Description
Read Channel	**rdch**	Causes a read of data stored in the addressed channel to be loaded into the selected general purpose register (GPR).
Write Channel	**wrch**	Causes data to be read from the selected GPR and stored in the addressed channel.
Read Channel Count	**rchcnt**	Causes the count associated with the addressed channel to be stored in the selected GPR.
Note: The Synergistic Processor Unit Instruction Set Architecture defines channels as 128 bits. The CBEA defines all channels a as 32 bits, and those 32 bits are in the preferred slot. See the *Synergistic Processor Unit Instruction Set Architecture* document for more information.		

Architecturally, SPU channels are defined as read only or write only; channels cannot be defined as both read and write. In addition to the access type, each channel can be defined as nonblocking or blocking. All blocking channels have an associated channel count. Nonblocking channels do not have an associated channel count; a read channel count instruction (**rchcnt**) that targets a nonblocking channel always returns a count of 1. Channels that are defined as blocking cause the SPU to stall when reading a channel with a channel count of 0, or when writing to a full channel (that is, a channel with a channel count of 0).

- Read—Means that data is always returned for a read channel instruction (**rdch**) that targets this channel.
- Write—Means that data is always accepted for a write channel instruction (**wrch**) that targets this channel.
- Read-blocking—Means that a read channel instruction (**rdch**) can target this channel. However, a read channel instruction (**rdch**) that targets a read-blocking channel only completes if the channel count is not 0. A channel count of 0 indicates that the channel is empty. Executing a read channel instruction (**rdch**) to a read-blocking channel with a count of 0 results in the SPU stalling until data is available in the channel.
- Write-blocking—Means that a write channel instruction (**wrch**) can target this channel. However, a write channel (**wrch**) instruction that targets a write-blocking channel only completes if the channel count is not 0. A channel count of 0 indicates that the channel is full. Executing a write channel (**wrch**) instruction to a write-blocking channel with a count of 0 results in the SPU stalling until an entry in the addressed channel becomes available.

Note: Issuing a channel instruction that is inappropriate for the definition of the channel results in an invalid channel instruction interrupt. For example, issuing a read channel instruction (**rdch**) to a channel defined as a write or write-blocking channel results in an invalid channel instruction interrupt.

Each blocking channel has a corresponding count (that is, depth), which indicates the number of outstanding operations that can be issued for that channel. The channel depth (that is, the maximum number of outstanding transfers) is implementation dependent. Software must initialize the channel counts when establishing a new context in the SPU, or when it resumes an existing context.

Table 9-2 lists the SPU channels grouped by function. Each Synergistic Processor Element (SPE) in the processor contains the full set of channels outlined in *Table 9-2* and an SPU channel access facility (see page 242). For the reader's convenience, *Table B-1 SPU Channel Map* on page 305 duplicates *Table 9-2* below.

Note: No reserved channels can be used for implementation-dependent functions.

Table 9-2. SPU Channel Map (Page 1 of 2)

Channel Number (Hexadecimal)	Channel Name	Description	Access Type
SPU Event Channels			
x'0'	SPU_RdEventStat	SPU Read Event Status Channel (see page 153). Read event status (with mask applied).	Read blocking
x'1'	SPU_WrEventMask	SPU Write Event Mask Channel (see page 157). Write event-status mask.	Write
x'2'	SPU_WrEventAck	SPU Write Event Acknowledgment Channel (see page 161). Write end-of-event processing.	Write
SPU Signal Notification Channels			
x'3'	SPU_RdSigNotify1	SPU Signal Notification 1 Channel (see page 143).	Read blocking
x'4'	SPU_RdSigNotify2	SPU Signal Notification 2 Channel (see page 144).	Read blocking
x'5'	Channel 5	Reserved	
x'6'	Channel 6	Reserved	
SPU Decrementer Channels			
x'7'	SPU_WrDec	SPU Write Decrementer Channel (see page 145).	Write
x'8'	SPU_RdDec	SPU Read Decrementer Channel (see page 146).	Read
MFC Multisource Synchronization Channels			
x'9'	MFC_WrMSSyncReq	MFC Write Multisource Synchronization Request Channel (see page 149).	Write blocking
SPU Reserved Channel			
x'A'	Channel 10	Reserved	
Mask Read Channels			
x'B'	SPU_RdEventMask	SPU Read Event Mask Channel (see page 159).	Read
x'C'	MFC_RdTagMask	MFC Read Tag-Group Query Mask Channel (see page 131).	Read
SPU State Management Channels			
x'D'	SPU_RdMachStat	SPU Read Machine Status Channel (see page 147).	Read
x'E'	SPU_WrSRR0	SPU Write State Save-and-Restore Channel (see page 148).	Write
x'F'	SPU_RdSRR0	SPU Read State Save-and-Restore Channel (see page 148).	Read
MFC SPU Command Parameter Channels			
x'10'	MFC_LSA	MFC Local Storage Address Channel (see page 121). Write local storage address command parameter.	Write

Table 9-2. SPU Channel Map (Page 2 of 2)

Channel Number (Hexadecimal)	Channel Name	Description	Access Type
x‘11’	MFC_EAH	MFC Effective Address High Channel (see page 124). Write high-order MFC SPU effective-address command parameter.	Write
x‘12’	MFC_EAL	MFC Effective Address Low or List Address Channel (see page 122). Write low-order MFC SPU effective-address command parameter.	Write
x‘13’	MFC_Size	MFC Transfer Size or List Size Channel (see page 120). Write MFC SPU transfer-size command parameter.	Write
x‘14’	MFC_TagID	MFC Command Tag Identification Channel (see page 119). Write MFC SPU tag identifier command parameter.	Write
x‘15’	MFC_Cmd MFC_ClassID	MFC Command Opcode Channel (see page 117). Write and enqueue MFC SPU command with associated class ID. MFC Class ID Channel (see page 118). Write and enqueue MFC SPU command with associated command opcode.	Write blocking
MFC Tag Status Channels			
x‘16’	MFC_WrTagMask	MFC Write Tag-Group Query Mask Channel (see page 129). Write tag mask.	Write
x‘17’	MFC_WrTagUpdate	MFC Write Tag Status Update Request Channel (see page 132). Write request for conditional or unconditional tag status update.	Write blocking
x‘18’	MFC_RdTagStat	MFC Read Tag-Group Status Channel (see page 133). Read tag status (with mask applied).	Read blocking
x‘19’	MFC_RdListStallStat	MFC Read List Stall-and-Notify Tag Status Channel (see page 135). Read MFC list stall-and-notify status.	Read blocking
x‘1A’	MFC_WrListStallAck	MFC Write List Stall-and-Notify Tag Acknowledgment Channel (see page 136). Write MFC list stall-and-notify acknowledgment.	Write
x‘1B’	MFC_RdAtomicStat	MFC Read Atomic Command Status Channel (see page 137). Read atomic command status.	Read blocking
SPU Mailboxes			
x‘1C’	SPU_WrOutMbox	SPU Write Outbound Mailbox Channel (see page 139). Write outbound SPU mailbox contents.	Write blocking
x‘1D’	SPU_RdInMbox	SPU Read Inbound Mailbox Channel (see page 141). Read inbound SPU mailbox contents.	Read blocking
x‘1E’	SPU_WrOutIntrMbox	SPU Write Outbound Interrupt Mailbox Channel (see page 140). Write SPU outbound interrupt mailbox contents.	Write blocking
x‘1F’ - x‘3F’	Channel 31 - Channel 63	Reserved	

9.1 MFC SPU Command Parameter Channels

The MFC proxy command parameter registers are described in *Section 8.1* on page 80. The channels used to queue an MFC SPU command to the MFC SPU command queue are described in the following sections:

- *Section 9.1.1 MFC Command Opcode Channel* beginning on page 117 describes using channel x‘15’ (lower bits).
- *Section 9.1.2 MFC Class ID Channel* beginning on page 118 describes using channel x‘15’ (upper bits).
- *Section 9.1.3 MFC Command Tag Identification Channel* beginning on page 119 describes using channel x‘14’.
- *Section 9.1.4 MFC Transfer Size or List Size Channel* beginning on page 120 describes using channel x‘13’.
- *Section 9.1.5 MFC Local Storage Address Channel* beginning on page 121 describes using channel x‘10’.
- *Section 9.1.6 MFC Effective Address Low or List Address Channel* beginning on page 122 describes using channel x‘12’.
- *Section 9.1.7 MFC Effective Address High Channel* beginning on page 124 describes using channel x‘11’.

The MFC SPU command parameter channels, except for the MFC Command Opcode Channel, are nonblocking. They do not have channel counts associated with them. A read channel count (**rchcnt**) instruction that targets these channels returns a count of 1.

The MFC Command Opcode Channel has a maximum count configured by hardware to the number of MFC SPU queue commands supported by the hardware. Software must initialize the MFC Command Opcode Channel count to the number of empty MFC SPU command queue slots supported by the implementation after power-on and after a purge of the MFC queue. This channel count must also be saved and restored on an SPE preemptive context switch.

Implementation Note:

An MFC SPU command must not be put into the MFC SPU command queue until a write channel (**wrch**) of the MFC SPU command to MFC Command Opcode Channel is executed by the SPU. The maximum count of the MFC Command Opcode Channel should be configured by hardware to the number of empty MFC SPU command slots supported by the implementation.

9.1.1 MFC Command Opcode Channel (MFC_Cmd)

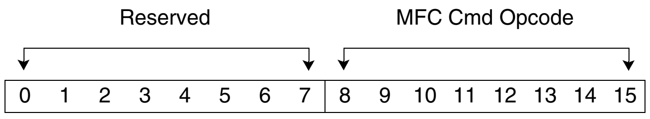
The MFC Command Opcode Channel determines the operation to be performed. The validity of this opcode is checked asynchronously to the instruction stream. If the MFC SPU command or any of the command parameters are invalid, MFC SPU command queue processing is suspended. If the interrupt is enabled, an invalid MFC SPU command interrupt is sent. For more information about interrupts, see *Section 21 Interrupt Facilities* beginning on page 261.

Software must avoid programming practices that place commands in the queue with forward dependencies on newer commands that are placed in the queue. Software with this type of dependency can create a deadlock based on the number of available slots in the MFC SPU command queue. In addition, while queue depth is implementation dependent, software must *not* be written to require a specific queue depth.

Software can determine the number of queue entries available in the MFC SPU command queue by issuing a read channel count (**rchcnt**) instruction that targets this channel. The value returned is the number of available queue slots. Software can use this value to avoid stalling the execution of an SPU program, which occurs when an MFC SPU command enqueue is attempted to a full queue.

Section 9.2 MFC SPU Command Issue Sequence beginning on page 125 describes the queuing sequence for MFC SPU commands.

Access Type	Write-blocking
Channel Number	x‘15’ (lower 16 bits)



Bits	Field Name	Description
0:7	Reserved	Should normally be set to zeros. If bit 0 is set to ‘1’, the opcode is reserved.
8:15	MFC Cmd Opcode	MFC SPU command opcode. See *Section 7 MFC Commands* beginning on page 55 for the MFC command opcodes.

Note: The MFC SPU command and the class ID parameters must be written to the MFC SPU command queue using a single channel instruction. The MFC SPU command parameter is the lower 16-bits of the 32-bit word. The upper 8 bits of this field are reserved. If the MSb (that is, bit 0 of this field) is set to ‘1’, the SPU command is reserved and can be used for an implementation-dependent function.

Programming Note:

The total number of queue slots is implementation dependent and varies between implementations. For portability of an application, the enqueue sequence for direct memory access (DMA) commands and the method to determine the number of queue slots available should be provided as a macro or library function that uses a configuration-dependent method for queueing commands.

9.1.2 MFC Class ID Channel (MFC_ClassID)

The MFC Class ID Channel is used to specify the replacement class ID and the transfer class ID for each MFC SPU command. Software can use these IDs to improve the overall performance of the system.

The transfer class ID (TclassID) lets an implementation optimize MFC SPU command transfers based on the characteristics of the storage location. The replacement class ID (RClassID) lets an implementation influence cache replacement. For more information on the replacement class ID, see *Section 19 Cache Replacement Management Facility* on page 255. The exact function and mapping of the replacement class ID and the transfer class ID are implementation dependent. For more information on setup and use of these IDs, see the specific implementation documentation.

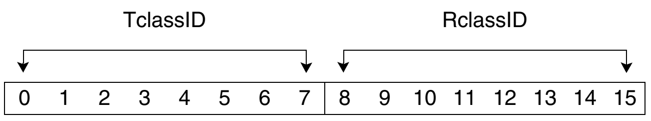
The MFC class ID performs the same function for commands that are issued to the MFC SPU command queue or to the MFC proxy command queue. Thus, its function is queue independent. For example, the RclassID affects which set in the translation lookaside buffer (TLB) is available to be replaced. The RClassID is a pointer into the replacement management table (RMT), which is shared between queues. Since the RMT is shared and the TLB is shared, the class ID is independent of the queue to which the command was issued. (A parameter such as the Tag parameter, on the other hand, is queue specific).

The MFC Class ID Channel controls resources that are associated with a specific SPE. It has no effect on resources associated with other SPEs or PPEs.

The contents of the class ID parameters are not persistent and must be written for each MFC SPU command enqueue sequence.

The validity of the class ID parameters is not verified. The number of class IDs supported is implementation dependent. The default class ID (x‘00’) is used for all undefined or invalid class IDs. An invalid class ID does not generate an exception.

Access Type	Write-blocking
Channel Number	x‘15’ (upper 16 bits)



Bits	Field Name	Description
0:7	TclassID	Transfer class identifier
8:15	RclassID	Replacement class identifier

Note: The MFC class ID parameters must be written to the MFC SPU queue along with the command parameter using a single channel instruction. The MFC SPU command parameter is the lower 16 bits of the 32-bit word. The MFC class ID parameter is the upper 16 bits of the 32-bit word.

9.1.3 MFC Command Tag Identification Channel (MFC_TagID)

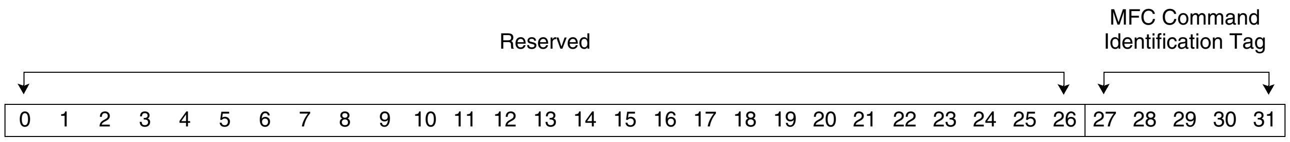
The MFC Command Tag Identification Channel is used to specify an identifier for each command, or for a group of commands. The identification tag is any value between x‘0’ and x‘1F’. Identification tags have a purely local scope in the hardware.

Any number of MFC SPU commands can be tagged with the same identification. MFC SPU commands tagged with the same identification are called a tag group. Tags are associated with commands written to a specific queue. Tags supplied to the MFC SPU command queue are independent of the tags supplied to the MFC proxy command queue.

The MFC Command Identification Tag parameter is not persistent and must be written for each MFC SPU command enqueue sequence.

The validity of this parameter is checked asynchronous to the instruction stream. If the upper bits (bits 0 through 26) are not set to zeros, MFC SPU command queue processing is suspended. If the interrupt is enabled, an invalid MFC SPU command interrupt is generated. For more information about interrupts, see *Section 21 Interrupt Facilities* beginning on page 261.

Access Type	Write
Channel Number	x‘14’



Bits	Field Name	Description
0:26	Reserved	Set to zeros.
27:31	MFC Command Tag Identification	MFC SPU command tag identification

9.1.4 MFC Transfer Size or List Size Channel (MFC_Size)

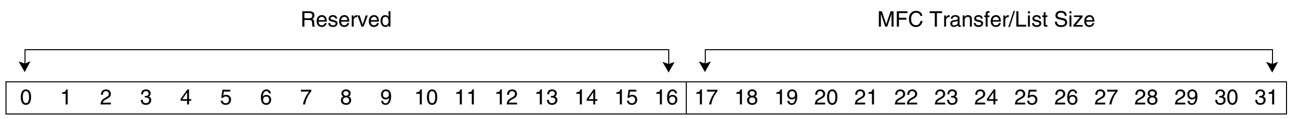
The MFC Transfer Size or List Size Channel is used to specify the size of the MFC transfer, or the size of the MFC DMA transfer list. The transfer size value can be 0, 1, 2, 4, 8, 16, or a multiple of 16 bytes to a maximum of 16 KB. The MFC transfer list size can have a value of 8, or of multiples of 8 up to a maximum of 16 KB.

The contents of the MFC Transfer Size or List Size Channel are not persistent and must be written for each MFC SPU command enqueue sequence.

The validity of this parameter is checked asynchronously to the instruction stream. If the size is invalid, the MFC SPU command queue processing is suspended. If the interrupt is enabled, an MFC DMA alignment interrupt is sent. For more information about interrupts, see *Section 21 Interrupt Facilities* beginning on page 261.

Access Type Write

Channel Number x‘13’



Bits	Field Name	Description
0:16	Reserved	Set to zero.
17:31	MFC Transfer/List Size	Valid MFC transfer size or list size: • 0 ≤ MFC transfer size ≤ 16 KB • 0 ≤ MFC list size ≤ 16 KB Allowable transfer sizes or list sizes follow: • 0, 1, 2, 4, and 8 bytes (naturally aligned), where the source and destination address *must* have the same 4 least-significant bits (see *Table 7-6 Command Errors and Alignment Errors* on page 61). • 16 bytes and multiples of 16 bytes up to 16 KB, where the source and destination addresses must be 16-byte (quadword) aligned. Allowable MFC list sizes follow: • Multiples of 8 bytes with a minimum size of 0 bytes and a maximum size of 16 KB for list size, where the list must start on an 8-byte (doubleword) boundary in local storage. This provides a list with from 0 to 2048 possible elements.

Programming Note:

Typically, systems are more efficient when they perform transfers that are the size of a cache line or multiple cache lines. Excessive use of small transfers (less than a cache line) can result in poor bus and memory bandwidth utilization. In addition, when transferring multiple quadwords, better performance can typically be achieved when the effective address and local storage address are both aligned within the cache line. For example, if the cache line size is 128-bytes, optimal performance can be achieved when bits 25 through 27 of the 32-bit local storage address are equal to bits 57 thorough 59 of the 64-bit effective address. The lower 4 bits of the effective address and local storage address must be zero for quadword transfers.

The method that an application uses to determine the cache line size of the processor is implementation dependent. For more information, see the specific implementation documentation and software interface specifications.

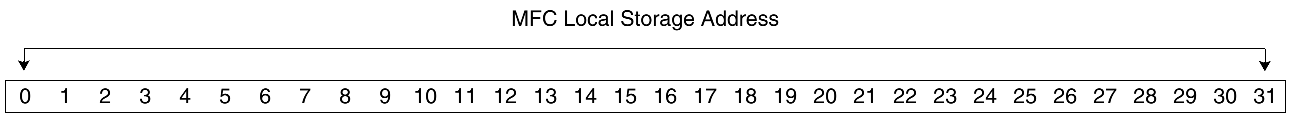
9.1.5 MFC Local Storage Address Channel (MFC_LSA)

The MFC Local Storage Address Channel is used to supply the SPU local storage address associated with an MFC SPU command to be queued. This address is used as the source or destination of the MFC transfer as defined in the MFC SPU command. For more information, see *Section 7 MFC Commands* beginning on page 55.

The contents of the MFC Local Storage Address Channel are not persistent and must be written for each MFC SPU command enqueue sequence.

The validity of this parameter is checked asynchronously to the instruction stream. To be considered aligned, the 4 least-significant bits of the local storage address must match the least-significant 4 bits of the effective address. If the address is unaligned, MFC SPU command queue processing is suspended. If the interrupt is enabled, an MFC DMA alignment interrupt is sent. For more information about interrupts, see *Section 21 Interrupt Facilities* beginning on page 261.

Access Type	Write
Channel Number	x‘10’



Bits	Field Name	Description
0:31	MFC Local Storage Address	MFC local storage address **Note:** The 4 least-significant bits of the local storage address must match the least-significant 4 bits of the effective address.

Programming Note:

Typically, systems are more efficient when they perform transfers of a cache line or multiple cache lines. Excessive use of small transfers (less than a cache line) can result in poor bus and memory bandwidth utilization. In addition, when transferring multiple quadwords, better performance can typically be achieved when the effective address and local storage address are both aligned within the cache line. For example, if the cache line size is 128 bytes, optimal performance can be achieved when bits 25 through 27 of the 32-bit local storage address are equal to bits 57 thorough 59 of the 64-bit effective address. The lower 4 bits of the effective address and local storage address must be zero for quadword transfers.

The method that an application uses to determine the cache line size of the processor is implementation dependent. For more information, see the specific implementation documentation and software interface specifications.

9.1.6 MFC Effective Address Low or List Address Channel (MFC_EAL)

The MFC Effective Address Low or List Address Channel is used to specify the lower 32 bits of the 64-bit effective address for the MFC SPU command, or the local storage pointer to the list elements for a list command. If translation is enabled in MFC State Register One (that is, MFC_SR1[R] is set to ‘1’), effective addresses are translated into real addresses by the memory management unit (MMU) address translation facility (for more information, see *PowerPC Architecture, Book III*). If translation is disabled, the 64-bit address formed by MFC_EAH || MFC_EAL must be a supported address within the real address space of main storage. Handling of unsupported addresses is implementation dependent. For more information about the real address limits of an implementation, see the specific implementation documentation.

The contents of the MFC Effective Address Low or List Address Channel are not persistent and must be written for each MFC SPU command enqueue sequence.

For transfer sizes less than 16 bytes, MFC_EAL bits 28 through 31 must provide natural alignment based on the transfer size. For transfer sizes of 16 bytes or greater, bits 28 through 31 must be zeros. In addition to these limitations, bits 28 through 31 must match bits 28 through 31 of the MFC_LSA. For list commands, bits 29 through 31 of the List Address must be zeros. If any of these conditions are not met, the MFC_EAL parameter is considered unaligned and is invalid.

The validity of this parameter is checked asynchronously to the instruction stream. If the address is invalid (for example, due to a segment fault, a mapping fault, a protection violation, or an alignment error), processing of the MFC SPU command queue is suspended. An interrupt, if enabled, is sent. Processing of other commands in the queue continues if possible, subject to any fence or barrier commands.

The following types of interrupts can be sent:

- If a segment fault occurs, an MFC data segment interrupt is sent.
- If a mapping fault or a page protection violation occurs, an MFC data-storage interrupt is sent.
- If the address is not aligned, a DMA alignment interrupt is sent.

For more information about interrupts, see *Section 21 Interrupt Facilities* beginning on page 261.

Notes:

- The validity of the effective address is checked during transfers. Partial transfers can be performed before an invalid address is encountered and the exception is generated.
- For optimal performance of transfers of 128 bytes or more, the source and destination transfer addresses should be 128-byte aligned (bits 25 through 31 set to 0).

Access Type	Write
Channel Number	x‘12’

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 local storage address of the MFC list. For transfer sizes less than 16 bytes, address bits 28 through 31 must provide natural alignment based on the transfer size. For transfer sizes of 16 bytes or greater, bits 28 through 31 must be 0. If translation is disabled (the effective address equals the real address), some higher-order bits must also be zero, depending on the amount of real memory in the system. For MFC list operations, this parameter contains the local storage address of the MFC list. In this case, the address must begin on an 8-byte boundary.

Programming Note:

Typically, systems are more efficient when they perform transfers of a cache line or multiple cache lines. Excessive use of small transfers (less than a cache line) can result in poor bus and memory bandwidth utilization. In addition, when transferring multiple quadwords, better performance can typically be achieved when the effective address and local storage address are both aligned within the cache line. For example, if the cache line size is 128 bytes, optimal performance can be achieved when bits 25 through 27 of the 32-bit local storage address are equal to bits 57 thorough 59 of the 64-bit effective address. The lower 4 bits of the effective address and local storage address must be zero for quadword transfers.

The method that an application uses to determine the cache line size of the processor is implementation dependent. For more information, see the specific implementation documentation and software interface specifications.

9.1.7 MFC Effective Address High Channel (MFC_EAH)

The MFC Effective Address High Channel is used to specify the upper 32 bits of the 64-bit effective address for the MFC SPU command. If translation is enabled in MFC State Register One (that is, MFC_SR1[R] is ‘1’), effective addresses are translated into real addresses by the MMU address translation facility (for more information, see *PowerPC Architecture, Book III*). If translation is disabled (the effective address equals the real address), the number of lower effective address bits that are valid is implementation dependent (for more information, see the specific implementation documentation).

The contents of the MFC Effective Address High Channel are not persistent and must be written for each MFC SPU command enqueue sequence.

This parameter is optional. If the effective address high (EAH) is not written, then the hardware sets the EAH parameter to zero (that is, the address is between 0 and 2^{32}-1).

The validity of this parameter is checked asynchronously to the instruction stream. If the address is invalid (for example, due to a segment fault, a mapping fault, or a protection violation), processing of the MFC SPU command queue is suspended. An interrupt, if enabled, is sent. Processing of other commands in the queue continues if possible, subject to any fence or barrier commands.

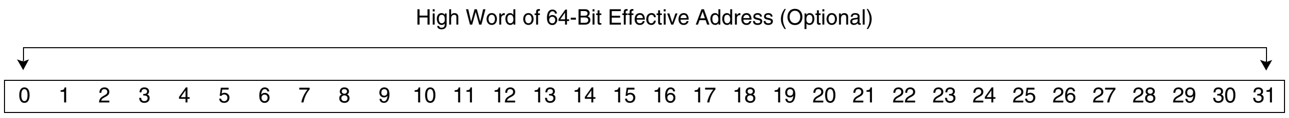
The following types of interrupts can be sent:

- If a segment fault occurs, an MFC data-segment interrupt is sent.
- If a mapping fault or a page protection violation occurs, an MFC data-storage interrupt is sent.

For more information about interrupts, see *Section 21 Interrupt Facilities* beginning on page 261.

Note: The validity of the effective address is checked during transfers. Partial transfers can be performed before an invalid address is encountered and the exception is generated.

Access Type	Write
Channel Number	x‘11’



Bits	Field Name	Description
0:31	High Word of 64-Bit Effective Address (Optional)	High word of the 64-bit effective address. • EAH is optional. If not written, zeros are used for the upper 32 bits of the 64-bit effective address. • If translation is disabled, the number of lower effective address bits that are valid is implementation specific. The remaining upper bits are required to be zero.

9.2 MFC SPU Command Issue Sequence

To queue an MFC SPU command from the SPU, the MFC SPU command parameters must first be written to the MFC SPU command parameter channels. The following command parameters can be written in any order, except that step 6 must always be done last:

1. Write the local storage address parameter (32 bits).
2. Write the effective address high parameter (upper 32 bits).[1]
3. Write the effective address low or the list address parameter (lower 32 bits).
4. Write the MFC SPU transfer or list size parameter (16 bits).
5. Write the MFC SPU command tag parameter (16 bits).
6. Write the MFC SPU command opcode and class ID parameter (32 bits).[2]

The MFC SPU command parameters are retained in the MFC SPU command parameter channels until the MFC processes a write of the MFC SPU command opcode and class ID parameter.

A write channel (**wrch**) instruction targeted to the MFC Command Opcode Channel and MFC Class ID Channel causes the parameters held in the MFC SPU command parameter channels to be sent to the MFC SPU command queue. The MFC SPU command parameters can be written in any order before the issue of the MFC SPU command itself. The values of the last parameters written to the MFC SPU command parameter channels are used in the enqueuing operation.

After an MFC SPU command has been queued, the values of the MFC parameters become invalid and must be respecified for the next MFC SPU command queuing request. Not specifying all of the required MFC parameters (that is, all the parameters except for the optional EAH) can result in the improper operation of the MFC SPU command queue.

The MFC SPU command parameter channels, except for the MFC Command Opcode Channel, are nonblocking. They do not have channel counts associated with them. A read channel count (**rchcnt**) instruction that targets these channels returns a count of 1.

The MFC Command Opcode Channel and MFC Class ID Channel have a maximum count configured by hardware to the number of MFC SPU queue commands supported by hardware. Software must initialize the channel count of the MFC Command Opcode Channel to the number of empty MFC SPU command queue slots supported by the implementation after power on and after a purge of the MFC SPU command queue. The channel count of the MFC Command Opcode Channel must also be saved and restored on an SPE preemptive context switch. (For more information, see the SPE context save sequence in the specific implementation documentation.)

1. This parameter is optional and is set to zero if not written.
2. This write channel (**wrch**) instruction stalls the SPU until there is room in the MFC queue for the command.

9.3 MFC Tag-Group Status Channels

This section describes how to determine if the MFC SPU commands for a tag group are complete. It also describes how to determine if a list command is stalled on a list element, which has its stall-and-notify flag set to a ‘1’. In addition, it describes how to subsequently restart the list command.

Each MFC SPU command is tagged with a 5-bit identifier (that is, the MFC command tag). The same identifier can be used for multiple MFC SPU commands. A set of commands in the same queue with the same identifier is called a tag group. Software can use this identifier to determine when a command or group of commands has completed. (That is, it can use the MFC command tag to check or to wait on the completion of all queued commands for each tag group.) In addition, software uses the MFC command tag to check or to wait on a list command, to reach an element with a stall-and-notify flag set, and to acknowledge the list element for resuming the list command. An interrupt can also be sent to a processor or device upon the completion of one or more tag groups if the interrupt is enabled by privileged software. For more information, see *Section 7.4 List Commands and List Elements* on page 63.

When the status returned by a channel read instruction that targets the MFC Read Tag-Group Status Channel (see page 133) indicates that a **put** command is complete, the local storage accesses are complete. The accesses are ordered with respect to the SPU. However, the main storage accesses might not be complete. The accesses are not ordered with respect to other processors and devices. For a **get** command, both the local storage and main storage accesses are complete and ordered with respect to other processors and devices.

9.3.1 Determining the Status of Tag Groups

Three basic procedures are supported to determine the status of the tag groups:

- Poll the MFC Read Tag-Group Status Channel
- Wait for a tag-group update, or wait for an event
- Interrupt on a tag-group status update event

The basic procedure for polling for the completion of an MFC SPU command, or for the completion of a group of MFC SPU commands, follows:

1. Clear any pending tag status update requests by performing the following steps:
   - Write ‘00’ to the MFC Write Tag Status Update Request Channel (see page 132).
   - Read the channel count associated with the MFC Write Tag Status Update Request Channel (see page 132) until a value of 1 is returned.
   - Read the MFC Read Tag-Group Status Channel (see page 133) and discard the tag status data.
2. Enable the tag groups of interest by writing the MFC Write Tag-Group Query Mask Channel (see page 129) with the appropriate mask data. This step is only needed if a new tag-group mask is required.
3. Request an immediate tag status update by writing the MFC Write Tag Status Update Request Channel (see page 132) with a value of 0.
4. Read the MFC Read Tag-Group Status Channel (see page 133). The data returned is the current status of each tag group with the tag-group mask applied.[1]
5. Repeat steps 3 and 4 until the tag group or the tag groups of interest are complete.

1. Performing a read of the MFC Tag Group Status Channel before issuing a request for a tag status update results in an SPU execution deadlock.

The basic procedure for waiting for a tag-group update or waiting for events (one or more tag-group completions) follows:

1. Clear any pending tag status update requests by performing the following steps:
   - Write ‘00’ to the MFC Write Tag Status Update Request Channel (see page 132).
   - Read the channel count associated with the MFC Write Tag Status Update Request Channel (see page 132), until a value of 1 is returned.
   - Read the MFC Read Tag-Group Status Channel (see page 133) and discard the tag status data.
2. Request a conditional tag status update by writing the MFC Write Tag Status Update Request Channel (see page 132), with a value of '01' or '10'. A value of '01' specifies that the completion of *any* enabled tag group results in a tag-group update. A value of '10' specifies that *all* enabled tag groups must complete to result in an SPU tag-group status update.
3. THEN EITHER:
   - Read from the MFC Read Tag-Group Status Channel (see page 133) to wait on the specific tag event specified in steps 1 and 2. This read stalls the execution of the SPU until the condition specified in step 2 is met.

OR

   - Read the count associated with the MFC Read Tag-Group Status Channel (see page 133) until a count of 1 is returned. Now, read from the MFC Read Tag-Group Status Channel (see page 133) to determine which tag group or tag groups are complete.

An alternative to waiting for or polling on a conditional tag event is to use the SPU event facility. This procedure is typically used when an application is waiting for one of multiple events to occur or can do other work while waiting for command completion. The procedure for using this facility follows:

1. Clear any pending tag status update requests by performing the following steps:
   - Write ‘00’ to the MFC Write Tag Status Update Request Channel (see page 132).
   - Read the channel count associated with the MFC Write Tag Status Update Request Channel (see page 132) until a value of 1 is returned.
   - Read the MFC Read Tag-Group Status Channel (see page 133) and discard the tag status data.

   After this step, select the tag group or tag groups.
2. Clear any pending tag status update events by writing (**wrch**) ‘1’ to the Tg bit of the SPU Write Event Acknowledgment Channel (see page 161).
3. Unmask the MFC Tag-Group Status Update Event (see page 163) by writing a '1' to the Tg bit of the SPU Write Event Mask Channel (see page 157).
4. THEN EITHER:
   - Read from the SPU Read Event Status Channel (see page 153) to wait for an enabled event to occur. This read stalls the execution of the SPU until an enabled event occurs.

OR

   - Read the count associated with the SPU Read Event Status Channel (see page 153) to poll or wait for the specific tag event until a count of 1 is returned. Read from the SPU Read Event Status Channel (see page 153) to determine which events occurred.
5. If an MFC tag-group status update event occurred, read (**rdch**) from the MFC Read Tag-Group Status Channel (see page 133) to determine which tag or tag groups caused the event.

9.3.2 Determining Command Completion

Three basic procedures are supported to determine if a list command has reached a list element with the stall-and-notify flag set:

- Poll the MFC Read List Stall-and-Notify Tag Status Channel (see page 135)
- Wait for an MFC list command stall-and-notify event (see page 163)
- Interrupt on MFC list command stall-and-notify event

The basic procedure for polling to determine if a list command has reached a list element with the stall-and-notify flag set follows:

1. Issue a list command that has a list element with the stall-and-notify flag set.
2. Read the count (**rchcnt**) associated with the MFC Read List Stall-and-Notify Tag Status Channel (see page 135) until a value of 1 is returned.
3. Read (**rdch**) the MFC Read List Stall-and-Notify Tag Status Channel (see page 135). The data returned is the current status of each tag group that has reached a list element with the stall-and-notify flag set since the last read of this channel.
4. Repeat steps 2 and 3 until the tag group or tag groups of interest have reached the list element with the stall-and-notify flag set. (The corresponding bits are set in the return data.)
5. Write (**wrch**) the MFC Write List Stall-and-Notify Tag Acknowledgment Channel (see page 136) with the tag-group number corresponding to the stalled tag group to resume the list command.

The basic procedure for waiting for a list command to reach a list element with the stall-and-notify flag set follows:

1. Issue a list command that has a list element with the stall-and-notify flag set.
2. Read (**rdch**) the MFC Read List Stall-and-Notify Tag Status Channel (see page 135). The data returned is the current status of each tag group that has reached a list element with the stall-and-notify flag set since the last read of this channel. This read stalls the SPU until a list command has reached a list element with the stall-and-notify flag set.
3. Repeat step 2 until the tag group or groups of interest have reached the list element with the stall-and-notify flag set. (The corresponding bits are set in the return data. Because the bits are reset for each read, software must perform the accumulation of the tag groups while waiting on multiple tag groups to stall.)
4. Write (**wrch**) the MFC Write List Stall-and-Notify Tag Acknowledgment Channel (see page 136) with the tag-group number corresponding to the stalled tag group to resume the list command.

An alternative to waiting for or polling on MFC Read List Stall-and-Notify Tag Status is to use the SPU event facility. This procedure is typically used when other work can be performed by the SPU program while the list command is executing.

The procedure for using this facility follows:

1. Clear any pending MFC list command stall-and-notify events by writing (**wrch**) a ‘1’ to the Sn bit of the SPU Write Event Acknowledgment Channel (see page 161).
2. Enable the MFC list command stall-and-notify event by writing a ‘1’ to the Sn bit of the SPU Write Event Mask Channel (see page 157).
3. Issue a list command that has a list element with the stall-and-notify flag set.

4. THEN EITHER:
   - Read (**rdch**) from the SPU Read Event Status Channel (see page 153) to wait for an enabled event to occur. This read stalls the execution of the SPU until an enabled event occurs.

OR:

   - Read the count (**rchcnt**) associated with the SPU Read Event Status Channel (see page 153) to poll for the specific tag event until a count of 1 is returned.
   - Read **(rdch**) from the SPU Read Event Status Channel (see page 153) to determine which events occurred.
5. If an MFC list stall-and-notify event occurred, read (**rdch**) from the MFC Read List Stall-and-Notify Tag Status Channel (see page 135) to determine which tag group or groups caused the event.
6. Repeat steps 4 and 5 until the tag group or groups of interest have reached the list element with the stall-and-notify flag set. (The corresponding bits are set in the return data. Because the bits are reset for each read, software must perform the accumulation of the tag groups while waiting for multiple tag groups to stall.)
7. Write (**wrch**) the MFC Write List Stall-and-Notify Tag Acknowledgment Channel (see page 136) with the tag-group number corresponding to the stalled tag group to resume the list command.

9.3.3 MFC Write Tag-Group Query Mask Channel (MFC_WrTagMask)

The MFC Write Tag-Group Query Mask Channel is used to select the tag groups to be included in the query or wait operations. 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 data provided by this channel is retained by the MFC until changed by a subsequent write channel (**wrch**) to this channel. Therefore, the data does not need to be respecified for each status query or wait. If this mask is modified by software when a query request is pending, the meaning of the results is ambiguous. A pending query request should always be cancelled before this mask is modified. A query request can be cancelled by writing a value of ‘0’ (that is, immediate update) to the MFC Write Tag Status Update Request Channel (see page 132).

The current contents of this channel can be accessed by reading (**rdch**) the MFC Read Tag-Group Query Mask Channel (see page 131).

This channel is nonblocking and does not have an associated count. If a read channel count (**rchcnt**) instruction targets this channel, the count is always returned as 1.

Access Type Write

Channel Number x‘16’

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 is not part of a query or wait-on-tag-event operation. 1 Tag group is part of a query or wait-on-tag-event operation.

9.3.4 MFC Read Tag-Group Query Mask Channel (MFC_RdTagMask)

The MFC Read Tag-Group Query Mask Channel is used to read the mask value that specifies the tag groups in the MFC SPU queue that are included in query or wait operations. Reading this channel always returns the current value in the MFC Read Tag-Group Query Mask Channel associated with the MFC SPU command queue.

This channel can be used by an SPE context save and restore operation, which avoids the need for software shadow copies of the value last written to the MFC Write Tag-Group Query Mask Channel.

This channel is nonblocking and does not have an associated count. If a read channel count (**rchcnt**) instruction targets this channel, the count is always returned as 1.

Access Type Read

Channel Number x‘C’

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 is not part of a query or wait-on-tag-event operation. 1 Tag group is part of a query or wait-on-tag-event operation.

9.3.5 MFC Write Tag Status Update Request Channel (MFC_WrTagUpdate)

The MFC Write Tag Status Update Request Channel is used to control when the MFC tag-group status is updated in the MFC Read Tag-Group Status Channel (see page 133). The MFC Write Tag-Group Query Mask Channel controls which tag groups participate in the update requests and therefore which tag groups influence the value read from the MFC Read Tag-Group Status Channel.

The MFC Write Tag Status Update Request Channel can specify when the status is updated:

- Updated immediately
- Updated when *any* enabled MFC tag-group has a “no operation outstanding” status, or
- Updated only when *all* enabled MFC tag groups have a “no operation outstanding” status.

A write channel (**wrch**) instruction to this channel must occur before a read channel (**rdch**) instruction from the MFC Read Tag-Group Status Channel occurs. An MFC write tag status update request should be performed after setting the tag-group mask and after issuing the 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.

Reading from the MFC Read Tag-Group Status Channel (see page 133) without first requesting a status update by writing to the MFC Write Tag Status Update Request Channel results in a software-induced dead-lock.

Completing the following steps can cancel a previous MFC write tag status update request:

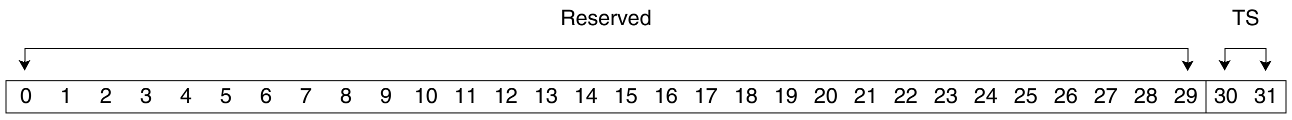
1. Issue an immediate update status request to the MFC Write Tag Status Update Request Channel.
2. Read the count associated with the MFC Write Tag Status Update Request Channel until a value of 1 is returned.
3. Read from the MFC Read Tag-Group Status Channel to determine if the operation is complete and to dis-card unwanted results.

Two conditional update requests without an intervening status read request result in the return of an unpre-dictable tag status. To avoid unpredictable results, software should pair requests for tag status updates with reads of the tag status, unless a request cancellation is performed with the immediate-update request.

Privileged software initializes the count for this channel to 1. The count for this channel is set to 0 when a write channel (**wrch**) instruction targets this channel. The count is set to 1 when the MFC receives the tag status update request.

This channel is write-blocking enabled with a maximum count of 1.

Access Type	Write-blocking
Channel Number	x‘17’



Bits	Field Name	Description
0:29	Reserved	Reserved.

Bits	Field Name	Description
30:31	TS	Tag-status update condition 00 Update immediately, unconditional. 01 Update tag status if or when *any* enabled tag group has “no outstanding operation” status. 10 Update tag status if or when *all* enabled tag groups have “no outstanding operation” status. 11 Reserved.

Implementation Note:

When an immediate request type command targets this channel, the MFC must not acknowledge the write channel (**wrch**) of the MFC Write Tag Status Update Request Channel until it has processed the command and updated the MFC Read Tag-Group Status Channel with the results. An immediate request type command must cancel any pending conditional request command. When the MFC has advised the hardware to check for the condition, the MFC acknowledges the write channel (**wrch**) of the MFC Write Tag Status Update Request Channel. When a conditional request type command targets this channel, acknowledging the write of this channel causes the channel count to change from 0 to 1.

9.3.6 MFC Read Tag-Group Status Channel (MFC_RdTagStat)

The MFC Read Tag-Group Status Channel contains the status of the tag groups from the last tag-group status update request. Only the status of the enabled tag groups at the time of the tag-group status update is valid. The bit positions that correspond to the tag groups that are disabled at the time of the tag-group status update are set to '0'.

When the status of a channel read instruction that targets the MFC Read Tag-Group Status Channel indicates that a **put** command is complete, the local storage accesses are complete. The accesses are ordered with respect to the SPU. However, the main storage accesses might not be complete. The accesses are not ordered with respect to other processors and devices. For a **get** command, both the local storage and main storage accesses are complete and ordered with respect to other processors and devices.

A write channel (**wrch**) instruction must target the MFC Write Tag Status Update Request Channel before reading from this channel. Failure to do so results in a software-induced deadlock condition. This is considered a programming error, and privileged software is required to remove the deadlock condition.

A read channel count (**rchcnt**) instruction that targets the MFC Read Tag-Group Status Channel returns 0 if the status is not yet available. It returns 1 if the status is available. This instruction can be used to avoid stalling the SPU when the MFC Read Tag-Group Status Channel is read.

Software initializes the count for this channel to a value of 0. This channel is read-blocking enabled, with a maximum count of 1.

Access Type Read-blocking

Channel Number x‘18’

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 1 Tag group has no outstanding operations (commands are complete) and was not disabled by the query mask. 0 Tag group has outstanding operations or has been disabled by the query mask.

9.3.7 MFC Read List Stall-and-Notify Tag Status Channel (MFC_RdListStallStat)

List elements for a list command contain a stall-and-notify flag. If the flag is set on a list element, the MFC stops executing the list command after completing (from the SPU's perspective) the transfer requested by this element. The flag sets the bit corresponding to the tag group of the list command in this channel. The count associated with this channel is also set to 1. A list command remains stalled until acknowledged by writing the tag value to the MFC Write List Stall-and-Notify Tag Acknowledgment Channel (see page 136).

Note: The MFC list stall-and-notify facility is useful when a program wants to be notified when MFC list execution has reached a specific point. This is also useful when an application wants to dynamically change list elements (transfer sizes or effective addresses) that follow the stalled list element. List elements can also be skipped by setting their transfer size to 0. Hardware is not allowed to prefetch list elements beyond a stall-and-notify element.

Privileged software should initialize the count of the MFC Read List Stall-and-Notify Tag Status Channel to zeros.

Software can determine which tag groups have commands that have stalled since the last read of this channel by reading the contents of this channel again. Issuing a read channel (**rdch**) instruction to this channel resets all bits to zeros, and sets the count corresponding to this channel to 0. Therefore, issuing a read channel (**rdch**) instruction with no outstanding list elements that contain a stall-and-notify flag set to '1' and no stalled commands results in a software-induced deadlock.

Issuing a read channel (**rdch**) instruction when no tag groups are stalled results in SPU execution stall until a list element with the stall-and-notify flag set is encountered.

Software can also read the count (**rchcnt**) associated with this channel or use the SPU event facility to determine when a list element is encountered with the stall-and-notify flag set.

A read channel count (**rchcnt**) instruction that targets the MFC Read List Stall-and-Notify Tag Status Channel returns 0 if there are no new stalled list commands since the last read of this channel.

This channel is read-blocking and has a maximum count of 1.

Access Type Read-blocking

Channel Number x'19'

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 1 Tag group has a list command that has stalled on an element with the stall-and-notify flag set. 0 Tag group has no list commands currently stalled.

Programming Note:

Programmers should avoid issuing multiple list commands that have the stall-and-notify flags set within the same tag group, unless subsequent commands have a tag-specific fence. If multiple list commands are issued with the same tag ID and without a tag-specific fence, software cannot determine which command or commands have reached the stall point. A programmer should always use a tag-specific fence or a barrier between list commands with the same tag ID that have a list element with a stall-and-notify flag set.

Implementation Notes:

1. A read channel (**rdch**) instruction resets all bits. If a bit is set at the same time as a read channel (**rdch**) instruction, that bit must remain set. The channel count must remain at 1 if the state of the bit is not reflected in the data returned for the read channel (**rdch**) instruction.
2. If hardware implements MFC list prefetching, it must not span an element that has the stall-and-notify flag set. Software can modify the list following a stall, before acknowledging and resuming list processing.

9.3.8 MFC Write List Stall-and-Notify Tag Acknowledgment Channel (MFC_WrListStallAck)

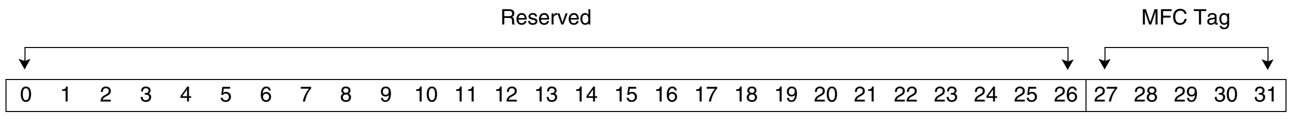
The MFC Write List Stall-and-Notify Tag Acknowledgment Channel is used to acknowledge a tag group containing list commands that are stalled on a list element with the stall-and-notify flag set. The tag group is acknowledged by writing the MFC tag identifier to this channel. After the write, all stalled list commands of the tag group for the identifier written to this channel are restarted.

Note: The MFC list stall-and-notify facility is useful when a program wants to be notified when MFC list execution has reached a specific point. This is also useful when an application wants to dynamically change list elements (transfer sizes or effective addresses) that follow the stalled list element. List elements can also be skipped by setting their transfer size to 0. Hardware is not allowed to prefetch list elements beyond a stall-and-notify element.

Acknowledging a tag group that is currently not stalled due to a stall-and-notify condition is undefined. Doing so can result in an invalid status in the MFC Read List Stall-and-Notify Tag Status Channel. For consistency, an implementation should treat this condition as a no-op.

This channel is nonblocking and does not have an associated count. Whenever a read channel count (**rchcnt**) instruction targets this channel, the count is always returned as 1.

Access Type	Write
Channel Number	x‘1A’



Bits	Field Name	Description
0:26	Reserved	Reserved.
27:31	MFC Tag	The tag can be any value between x‘0’ and x‘1F’.

9.4 MFC Read Atomic Command Status Channel (MFC_RdAtomicStat)

The MFC Read Atomic Command Status Channel contains the status of the last completed immediate MFC atomic update command (**getllar, putllc**, or **putlluc**). Issuing a read channel (**rdch**) instruction to this channel before issuing an immediate atomic command results in a software-induced deadlock.

Note: This channel does not provide any status for the queued **putqlluc** command.

Software can read the channel count (**rchcnt**) associated with this channel to determine if an immediate MFC atomic update command has completed.

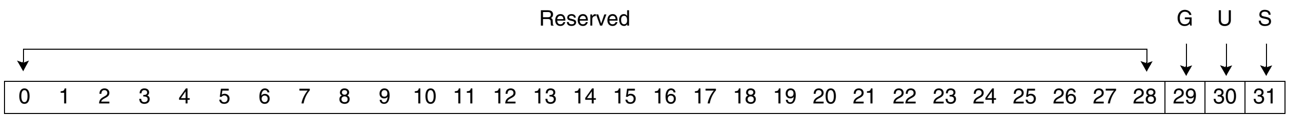
- If a value of 0 is returned, the immediate MFC atomic update command has not completed.
- If a value of 1 is returned, the immediate MFC atomic update command has completed, and the status is available by reading (**rdch**) this channel.

A read (**rdch**) from the MFC Read Atomic Command Status Channel should always follow an immediate MFC atomic update command. Performing multiple MFC atomic update commands without an intervening read from the MFC Read Atomic Command Status Channel results in an incorrect status.

Privileged software should initialize the count of this channel to 0. This channel is read-blocking with a maximum count of 1. The contents of this channel are cleared when read.

Completion of a subsequent immediate MFC atomic update command overwrites the status of an earlier MFC atomic update command.

Access Type	Read-blocking
Channel Number	x‘1B’



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.

9.5 SPU Mailbox Channels

This section describes the following channels, which are part of the mailbox facility (see page 101):

- The SPU Write Outbound Mailbox Channel (see page 139)
- The SPU Write Outbound Interrupt Mailbox Channel (see page 140)
- The SPU Read Inbound Mailbox Channel (see page 141)

The SPU mailbox channels are defined as blocking. That is, they stall the SPU when a channel is full (write-blocking) or when data is not available (read-blocking). The blocking method of a channel is very beneficial for power savings when an application has no other work to perform.

However, accessing these channels causes the SPU to stall for an indefinite period of time. Software can avoid stalling the SPU by using the SPU event facility (see page 150) or by reading the channel count associated with the mailbox channel.

9.5.1 SPU Write Outbound Mailbox Channel (SPU_WrOutMbox)

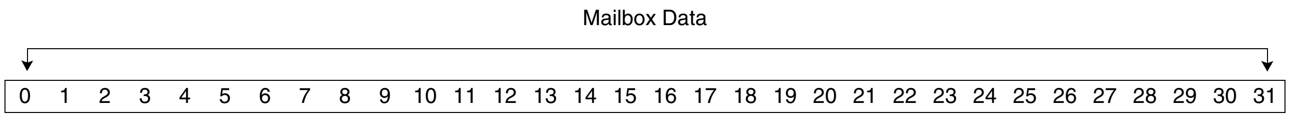
A write channel (**wrch**) to this channel writes data to the SPU write outbound mailbox queue. The data written to this channel by the SPU is available for a memory-mapped I/O (MMIO) read of the SPU Outbound Mailbox Register (see page 102).

A write channel (**wrch**) to this channel also causes the associated channel count to be decremented by 1. Writing to a full SPU write outbound mailbox queue causes SPU execution to stall until the SPU Outbound Mailbox Register (see page 102) is read, freeing up a location in the SPU write outbound mailbox queue. To avoid the stall condition, the channel count associated with this channel can be read to ensure there is a slot in the SPU write outbound mailbox queue before issuing the channel write. Alternatively, the SPU outbound mailbox available event can be used to signal the availability of a slot in the SPU write outbound mailbox queue, if it was determined to be full.

When the SPU write outbound mailbox queue is full, a read of the channel count associated with this channel returns a value of 0. A nonzero value indicates the number of 32-bit words free in the SPU write outbound mailbox queue.

Privileged software initializes the count of this channel to the depth of the SPU write outbound mailbox queue. This channel is write-blocking. The maximum count for this channel is implementation dependent. It should be the depth (that is, the number of available slots) of the SPU write outbound mailbox queue.

Access Type	Write-blocking
Channel Number	x'1C'



Bits	Field Name	Description
0:31	Mailbox Data	Application-specific mailbox data Each application can uniquely define the mailbox data.

Implementation Note:

The MFC must not acknowledge a write channel (**wrch**) instruction to the SPU Write Outbound Mailbox Channel until a processor or device has read the contents of the mailbox.

9.5.2 SPU Write Outbound Interrupt Mailbox Channel (SPU_WrOutIntrMbox)

A write channel (**wrch**) instruction to this channel writes data to the SPU write outbound interrupt mailbox queue. The data written to this channel by the SPU is made available to an MMIO read of the SPU Outbound Interrupt Mailbox Register (see page 237). (This register is located in the privilege 2 area of an SPE main storage address domain.)

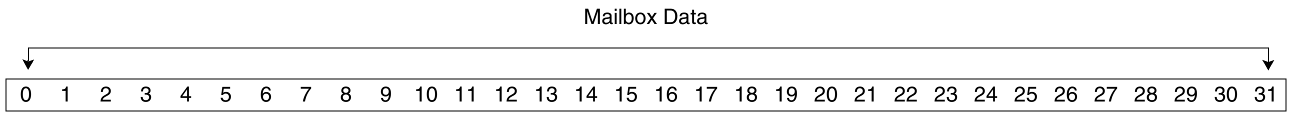
A write channel (**wrch**) instruction to this channel also causes the associated count to be decremented by 1. Writing to a full SPU write outbound interrupt mailbox queue causes SPU execution to stall until the SPU Outbound Interrupt Mailbox Register (see page 237) is read, freeing up a location in the SPU write outbound interrupt mailbox queue. To avoid a stall condition, the channel count associated with this channel can be read to ensure there is a slot in the SPU write outbound interrupt mailbox queue before issuing the channel write. Alternatively, the SPU outbound interrupt mailbox available event can be used to signal the availability of a slot in the SPU write outbound interrupt mailbox queue, if it was previously full.

A write channel (**wrch**) instruction that targets the SPU Write Outbound Interrupt Mailbox Channel also causes an interrupt to be sent to a processor or other device. There is no ordering of the interrupt and previously-issued MFC SPU commands. For more information, see *Section 21 Interrupt Facilities* beginning on page 261.

When the SPU write outbound interrupt mailbox queue is full, a read of the count associated with this channel returns a value of 0. A nonzero count value indicates the number of 32-bit words free in this queue.

Privileged software initializes the count of this channel to the depth of the SPU write outbound interrupt mailbox queue. This channel is write-blocking. The maximum count for this channel is implementation dependent. It should be the depth (that is, the number of available slots) of the SPU write outbound interrupt mailbox queue.

Access Type	Write-blocking
Channel Number	x‘1E’



Bits	Field Name	Description
0:31	Mailbox Data	Application-specific mailbox data Each application can uniquely define the mailbox data.

Implementation Note:

The MFC must not acknowledge a write channel (**wrch**) instruction to the SPU Write Outbound Interrupt Mailbox Channel until a processor or other device has read the contents of the mailbox.

9.5.3 SPU Read Inbound Mailbox Channel (SPU_RdInMbox)

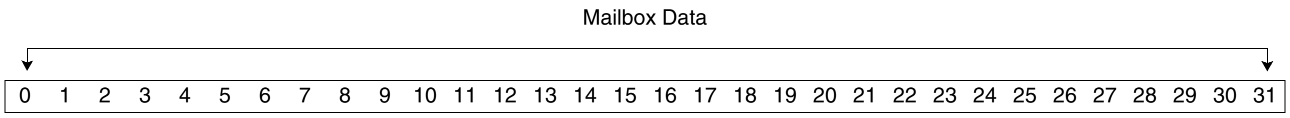
A read from this channel returns the next data in the SPU read inbound mailbox queue. Data is placed in the SPU read inbound mailbox queue by a processor or device issuing a write to the SPU Inbound Mailbox Register (see page 103).

Reading from the SPU Read Inbound Mailbox Channel also causes the associated count to be decremented by 1. Reading an empty mailbox causes SPU execution to stall until the SPU Inbound Mailbox Register (see page 103) is written, placing a data item in the SPU read inbound mailbox queue. To avoid the stall condition, the channel count associated with this channel can be read to ensure there is data in the SPU read inbound mailbox queue before issuing the channel read. Alternatively, the SPU inbound mailbox available event can be used to signal the availability of data in the SPU read inbound mailbox queue.

When the mailbox is empty, reading the channel count (**rchcnt**) returns a value of 0. If the result of the **rchcnt** is nonzero, then the mailbox contains information that has been written by a PPE but has not been read by a SPU.

Privileged software initializes the channel count of the SPU Read Inbound Mailbox Channel to 0. The maximum count for this channel is implementation dependent. This channel is read-blocking.

Access Type	Read-blocking
Channel Number	x‘1D’



Bits	Field Name	Description
0:31	Mailbox Data	Application-specific mailbox data Each application can uniquely define the mailbox data.

9.6 SPU Signalling Channels

These channels are the SPU side of the SPU signal notification facility (see page 105). They are used to read signals from other processors and other devices in the system. The signalling channels are configured as read-blocking with a maximum count of 1.

When a read channel (**rdch**) instruction targets one of these channels and the associated channel count is 1, the current contents of the channel and the associated count are reset to 0. When a read channel (**rdch**) instruction targets one of these channels, and the channel count is 0, the SPU stalls until a processor or device performs an MMIO write to the associated register.

Implementation Note:

When a signalling event coincides with a read of the signalling channel, hardware must ensure that the proper state of the channel is maintained. In overwrite mode, hardware can either return the data associated with the signalling event or the current contents of the channel (if the count is 1) for the read channel instruction. For logical OR mode, hardware can either return the data associated with the signalling event ORed with the current contents of the channel or the current contents of the channel. If the signalling event data or the logical OR of the event data and the current contents is returned, the channel count must end up as a 0. If the current contents are returned, the channel count must end up as 1.

9.6.1 SPU Signal Notification 1 Channel (SPU_RdSigNotify1)

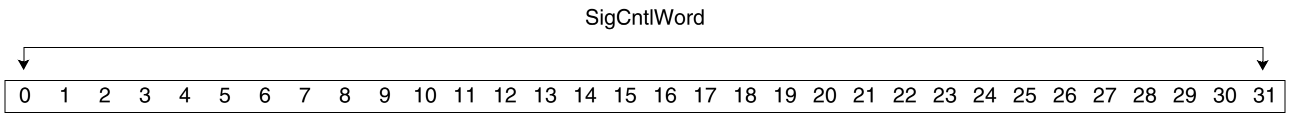
A read channel (**rdch**) instruction that targets the SPU Signal Notification 1 Channel returns the 32-bit value of the Signal-Control Word field. It atomically resets any bits that were set when read. If no signals are pending, a read from this channel stalls the SPU until a signal is issued.

If no signals are pending, a read channel count (**rchcnt**) instruction to this channel returns 0. If unread signals are pending, it returns 1.

Privileged software initializes the count for this channel to a value of 0. This channel is read-blocking enabled with a maximum count of 1.

Access Type Read-blocking

Channel Number x‘3’



Bits	Field Name	Description
0:31	SigCntlWord	Signal-control word. The application defines the data. It can either be ORed with the previous value, or it can be overwritten.

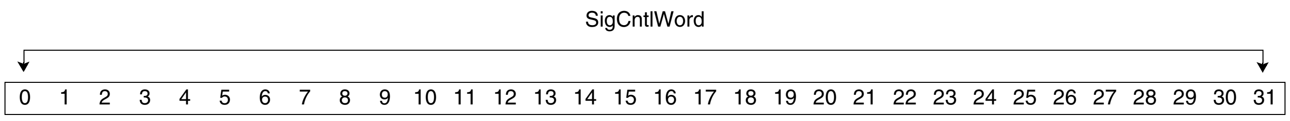
9.6.2 SPU Signal Notification 2 Channel (SPU_RdSigNotify2)

A read channel (**rdch**) instruction that targets the SPU Signal Notification 2 Channel returns the 32-bit value of the Signal-Control Word field. It atomically resets any bits that were set when read. If no signals are pending, a read from this channel stalls the SPU until a signal is issued.

A read channel count (**rchcnt**) instruction that targets this channel returns 0 if no signals are pending. It returns 1 if unread signals are pending.

Privileged software initializes the count for this channel to a value of 0. This channel is read-blocking enabled with a maximum count of 1.

Access Type	Read-blocking
Channel Number	x'4'



Bits	Field Name	Description
0:31	SigCntlWord	Signal-control word. The application defines the data. It can either be ORed with the previous value, or it can be over-written.

9.7 SPU Decrementer

Each SPU contains a 32-bit decrementer. It is enabled in the MFC Control Register (see page 233) when MFC_CNTL[Dh] is set to '0'. The SPU decrementer starts when a **wrch** instruction targets the SPU Write Decrementer Channel. The decrementer is stopped by following the procedure described in *Section 9.7.1*, or when MFC_CNTL[Dh] is set to '1'.

The current running status of the decrementer is available in the MFC Control Register (that is, MFC_CNTL[Ds]). A decrementer event does not need to be pending for the decrementer to be stopped.

Note: The requirement for a constant rate for the time base and the decrementer is an additional requirement beyond the PowerPC Architecture. However, all SPU decrementers are required to run at the same rate as the PPE decrementer, but there is no requirement that the decrementers be synchronized.

Two channels are assigned to manage the decrementer: one to set the decrementer value and one to read the current contents of the decrementer. A decrementer event occurs when the most-significant bit (bit 0) changes from a ‘0’ to a ‘1’.

9.7.1 SPU Write Decrementer Channel (SPU_WrDec)

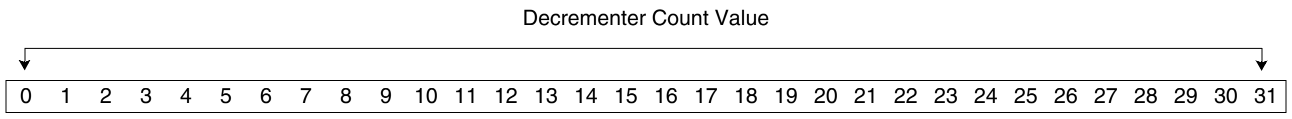
The SPU Write Decrementer Channel is used to load a 32-bit value to the decrementer. The value loaded into the decrementer determines the lapsed time between the write channel (**wrch**) instruction and the decrementer event. The event occurs when the most-significant bit of the decrementer changes from a ‘0’ to a ‘1’. 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.

For the state of the decrementer to be properly saved and restored, the decrementer must be stopped before changing the decrementer value. The following procedure sets a new decrementer value.

1. Write to the SPU Write Event Mask Channel (see page 157) to disable the decrementer event.
2. Write to the SPU Write Event Acknowledgment Channel (see page 161) to acknowledge any pending events and to stop the decrementer. The decrementer is stopped because the decrementer event has been disabled in step 1.
3. Write to the SPU Write Decrementer Channel to set a new decrementer count value. (Note: The decrementer is started because step 2 stopped the decrementer.)
4. Write to the SPU Write Event Mask Channel (see page 157) to enable the decrementer event.
5. Wait for the timer to expire.

This channel is nonblocking and does not have an associated count. Whenever a read channel count (**rchcnt**) instruction targets this channel, the count is always returned as 1.

Access Type	Write
Channel Number	x‘7’



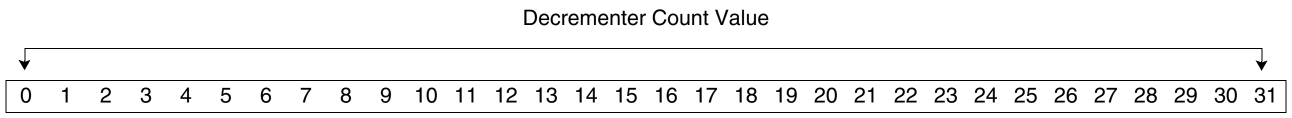
Bits	Field Name	Description
0:31	Decrementer Count Value	Decrementer count value.

9.7.2 SPU Read Decrementer Channel (SPU_RdDec)

The SPU Read Decrementer Channel is used to read the current value of the 32-bit decrementer. Reading the decrementer count has no effect on the accuracy of the decrementer. Successive reads of the decrementer can return the same value.

This channel is nonblocking and does not have an associated count. Whenever a read channel count (**rchcnt**) instruction targets this channel, the count is always returned as 1.

Access Type	Read
Channel Number	x‘8’



Bits	Field Name	Description
0:31	Decrementer Count Value	Decrementer count value.

9.8 SPU Read Machine Status Channel (SPU_RdMachStat)

The SPU Read Machine Status Channel contains the current SPU machine status information. This channel contains two status bits: the isolation status and the SPU interrupt enable status. The isolation status reflects the current operating state of the SPU, isolated or nonisolated. For more on SPU isolation, see *Section 11 SPU Isolation Facility* beginning on page 183.

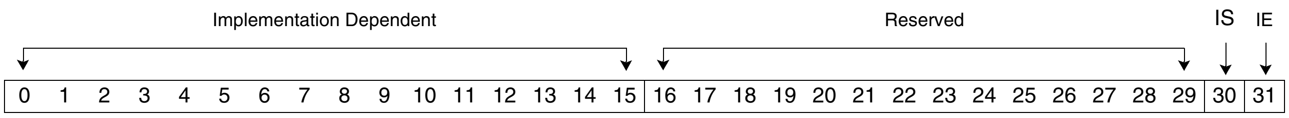
The SPU interrupt enable status reflects the current state of the SPU interrupt enable. If an interrupt is enabled and any enabled SPU event is present, an SPU interrupt is generated. For more information about SPU events, see *Section 9.11 SPU Event Facility* on page 150.

For more information about the processing of SPU interrupts, see the *Synergistic Processor Unit Instruction Set Architecture* document.

This channel is nonblocking and does not have an associated count. Whenever a read channel count (**rchcnt**) instruction targets this channel, the count is always returned as 1.

Access Type Read

Channel Number x‘D’



Bits	Field Name	Description
0:15	Implementation Dependent	
16:29	Reserved	Set to zeros.
30	IS	Isolation status. 0 The SPU is operating in a nonisolated state. 1 The SPU is operating in an isolated state.
31	IE	SPU interrupt enable status. Interrupts can be enabled by setting the SPU_NPC[IE] bit to '1' while the SPU is stopped or by an SPU instruction. See the *Synergistic Processor Unit Instruction Set Architecture* document for more information about how to enable interrupts. 0 SPU interrupts disabled. 1 SPU interrupt enabled.

9.9 SPU Interrupt-Related Channels

9.9.1 SPU Write State Save-and-Restore Channel (SPU_WrSRR0)

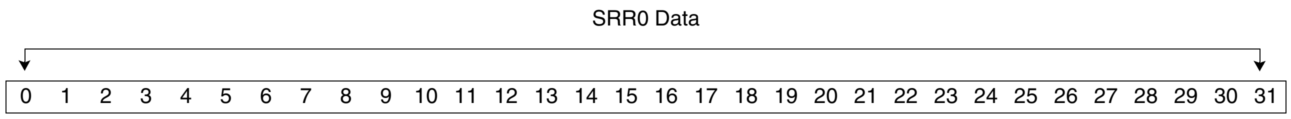
A write to this channel updates the contents of the State Save and Restore Register (SRR0) in the SPU. For more information, see the *Synergistic Processor Unit Instruction Set Architecture* document. A write to this channel is typically used to restore interrupt-state information when nested interrupts are supported.

This channel should not be written when SPU interrupts are enabled. Doing so can result in the contents of SRR0 being indeterminate. The **sync.c** instruction (a channel form of the **sync** instruction) must be issued after writing this channel and before the execution of instructions that are dependent upon the SRR0 contents.

This channel is nonblocking and does not have an associated count. Whenever a read channel count (**rchcnt**) instruction targets this channel, the count is always returned as 1.

Access Type Write

Channel Number x‘E’



Bits	Field Name	Description
0:31	SRR0 Data	State save/restore register 0 data.

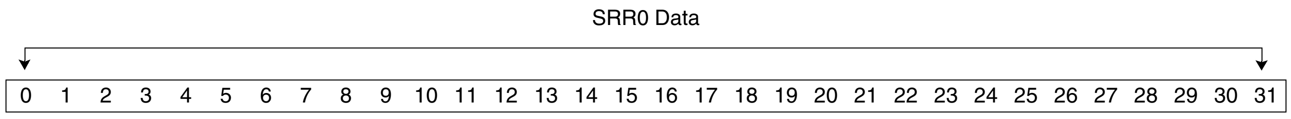
9.9.2 SPU Read State Save-and-Restore Channel (SPU_RdSRR0)

A read of this channel returns the contents of the State Save and Restore Register (SRR0) in the SPU. A read of this channel is typically used to save interrupt-state information when nested interrupts are supported. (For more information, see the *Synergistic Processor Unit Instruction Set Architecture* document.)

This channel is nonblocking and does not have an associated count. Whenever a read channel count (**rchcnt**) instruction targets this channel, the count is always returned as 1.

Access Type Read

Channel Number x‘F’



Bits	Field Name	Description
0:31	SRR0 Data	State save/restore register 0 data.

9.10 MFC Write Multisource Synchronization Request Channel (MFC_WrMSSyncReq)

The MFC Write Multisource Synchronization Request Channel is part of the MFC multisource synchronization facility (see page 108). It causes the MFC to start tracking outstanding transfers sent to the associated MFC. When the synchronization requested by a write of this channel is complete, the channel count is set back to 1. The data written to this channel is ignored; however, software should write a value of 0 for compatibility with future enhancements.

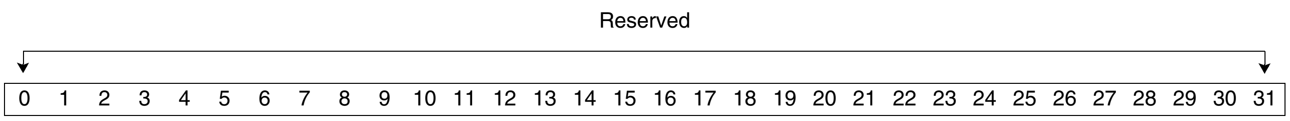
A second write to this channel results in the SPU being stalled until the outstanding transfers being tracked by the first write are complete.

To use the MFC Write Multisource Synchronization Request Channel, a program must perform the following steps:

1. Write to the MFC Write Multisource Synchronization Request Channel.
2. Wait for the MFC Write Multisource Synchronization Request Channel to become available (that is, when the channel count is set back to 1).

Software initializes the count for this channel to a value of 1. This channel is write-blocking enabled with a maximum count of 1.

Access Type	Write-blocking
Channel Number	x‘9’



Bits	Field Name	Description
0:31	Reserved	Reserved.

9.11 SPU Event Facility

An SPU program can monitor events by using the following channels:

- SPU Read Event Status Channel (see page 153)
- SPU Write Event Mask Channel (see page 157)
- SPU Read Event Mask Channel (see page 159)
- SPU Write Event Acknowledgment Channel (see page 161)

The SPU Read Event Status Channel contains the status of all events enabled in the SPU Write Event Mask Channel. The SPU Write Event Acknowledgment Channel is used to reset the status of an event, which typically indicates that the event has been processed or recorded by the SPU program. If no enabled events are present, reading from the SPU Read Event Status Channel stalls the SPU program. While individual events have a similar methods for stalling the SPU program, if the event has not occurred, the SPU event facility provides software with a method to look for multiple events and to cause an interrupt of the SPU program.

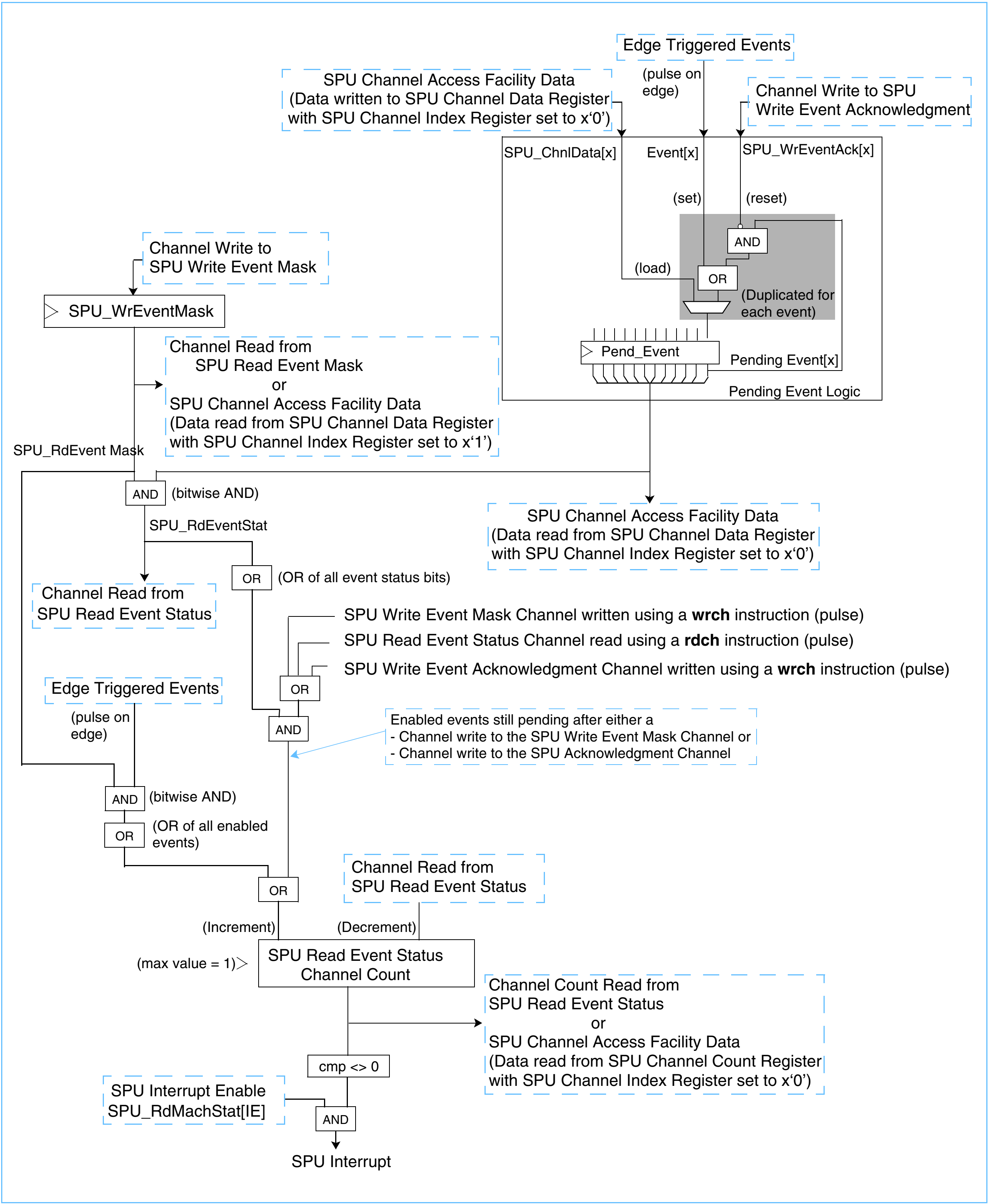
Several events can be monitored. For more information about these events, see *Section 9.12 SPU Event Definitions* beginning on page 163.

Figure 9-1 on page 151 is a logical representation of the function of each channel that supports the SPU event facility.

Programming Note:

When using the SPU event facility, software should avoid any conditions that might prevent an expected event from being presented to an SPU. An error condition, for example, might prevent an event from being presented. Software should also be designed to handle this situation if it does occur. Otherwise, the SPU could stall indefinitely while waiting for an event. To break out of an indefinite wait, the software design should include some form of a watchdog timer (contact your system provider about the preferred form of the watchdog timer). The specific actions software takes if the watchdog timer expires before the expected event takes place are implementation dependent. See *Section 9.12.10* on page 170 for a description of a specific condition that can occur when using the lock line reservation lost event.

Figure 9-1. Logical Representation of SPU Event Support


Edge Triggered Events
(pulse on edge)
SPU Channel Access Facility Data (Data written to SPU Channel Data Register with SPU Channel Index Register set to x‘0’)
Channel Write to SPU Write Event Acknowledgment
SPU_ChnlData[x]
Event[x]
SPU_WrEventAck[x]
(set)
(reset)
AND
(load)
OR
(Duplicated for each event)
Pend_Event
Pending Event[x]
Pending Event Logic
Channel Write to SPU Write Event Mask
SPU_WrEventMask
Channel Read from SPU Read Event Mask or SPU Channel Access Facility Data (Data read from SPU Channel Data Register with SPU Channel Index Register set to x‘1’)
SPU_RdEvent Mask
AND
(bitwise AND)
SPU_RdEventStat
SPU Channel Access Facility Data (Data read from SPU Channel Data Register with SPU Channel Index Register set to x‘0’)
OR
(OR of all event status bits)
Channel Read from SPU Read Event Status
SPU Write Event Mask Channel written using a **wrch** instruction (pulse)
SPU Read Event Status Channel read using a **rdch** instruction (pulse)
SPU Write Event Acknowledgment Channel written using a **wrch** instruction (pulse)
Edge Triggered Events
OR
(pulse on edge)
AND
Enabled events still pending after either a
- Channel write to the SPU Write Event Mask Channel or
- Channel write to the SPU Acknowledgment Channel
AND
(bitwise AND)
OR
(OR of all enabled events)
Channel Read from SPU Read Event Status
OR
(Increment)
(Decrement)
(max value = 1)
SPU Read Event Status Channel Count
Channel Count Read from SPU Read Event Status or SPU Channel Access Facility Data (Data read from SPU Channel Count Register with SPU Channel Index Register set to x‘0’)
cmp <> 0
SPU Interrupt Enable SPU_RdMachStat[IE]
AND
SPU Interrupt

As illustrated in *Figure 9-1* on page 151, an edge-triggered event sets a corresponding bit in the SPU Pending Event Register to a ‘1’. Events in the SPU Pending Event Register are acknowledged, or reset, by writing a ‘1’ to the corresponding bit in the SPU Write Event Acknowledgment Channel (see page 161) using a write channel instruction.

The SPU Pending Event Register (Pend_Event) is an internal register. The format of the SPU Pending Event Register is the same as the SPU Read Event Status Channel (see page 153). The SPU Pending Event Register can be read using the SPU channel access facility (see page 242). Reading the SPU Read Event Status Channel with the read channel (**rdch**) instruction returns the value of the SPU Pending Event Register logically ANDed with the value in the SPU Write Event Mask Channel (see page 157). This function provides an SPU program with only the status of the enabled events. The SPU Pending Event Register, however, allows privileged software to see all the events that have occurred. Access to all events is required for an SPE context save and restore operation.

The contents of the SPU Read Event Status Channel (see page 153) change when the SPU Write Event Mask Channel is written with a new value, or when a new event is recorded in the SPU Pending Event Register. Any transition of a bit from ‘0’ to ‘1’ in the SPU Read Event Status Channel increments the SPU Read Event Status Channel count by 1. The count also increments if an event is still set in the SPU Read Event Status Channel after a write to the SPU Write Event Acknowledgment Channel (see page 161). The count is decremented by 1 when the SPU Read Event Status Channel is read using a read channel (**rdch**) instruction. The count saturates at a value of 1, and is not decremented below a value of 0.

When the SPU Read Event Status Channel count is nonzero, an interrupt condition is sent to the SPU, if the interrupt is enabled. The *Synergistic Processor Unit Instruction Set Architecture* document describes enabling, disabling, and processing an interrupt.

Programming Note:

Software should acknowledge all events to be processed before processing the events. For example, a pending SPU inbound mailbox available event should be acknowledged before reading the SPU Read Inbound Mailbox Channel. If an SPU inbound mailbox available event and an SPU signal notification 1 available event are to be processed, both events should be acknowledged in the same write to the SPU Write Event Acknowledgment Channel before reading the SPU Signal Notification 1 Channel. If each event was acknowledged separately, the event status count would be incremented unnecessarily and a phantom interrupt would be sent, if interrupts are enabled (that is, SPU_RdMachStat[IE]).

9.11.1 SPU Read Event Status Channel (SPU_RdEventStat)

The SPU Read Event Status Channel contains the current status of all events enabled by the SPU Write Event Mask Channel (see page 157) at the time this channel is read. If the SPU Write Event Mask Channel specifies that an event is not part of the query, then its corresponding position is ‘0’ in the reported status.

A read from the SPU Read Event Status Channel when it has a channel count of 0 results in an SPU stall; thereby providing a “wait on event” function. A read from this channel when it has a channel count of 1 returns the status of any enabled, pending events and sets the channel count to 0. The channel count is set to 1 for the following conditions:

- An event occurs, and the corresponding mask is ‘1’ in the SPU Write Event Mask Channel (see page 157).
- The SPU Write Event Mask Channel is written with a ‘1’ in a bit position that corresponds to a ‘1’ in the SPU Pending Event Register.
- Enabled events are pending after a write of the SPU Write Event Acknowledgment Channel (see page 161).
- Privileged software sets the channel count to 1 using the SPU channel access facility (see page 242).

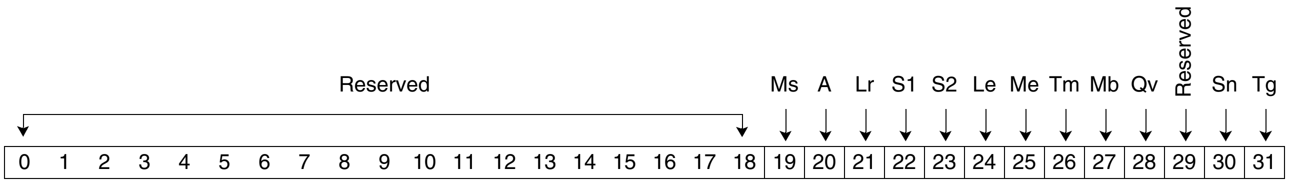
If no enabled events have occurred, a read channel count (**rchcnt**) instruction that targets the SPU Read Event Status Channel returns zeros. A read channel count (**rchcnt**) instruction can be used to avoid stalling the SPU when reading the event status from the SPU Read Event Status Channel. The channel count of the SPU Read Event Status Channel is also used as the condition in the branch indirect and set link if external data **(bisled**) instruction. If the SPU Read Event Status Channel count is 0, the branch is not taken. For more information about the **bisled** instruction, see the *Synergistic Processor Unit Instruction Set Architecture* document.

Privileged software must initialize the count value of the SPU Read Event Status Channel to 0.[1] The channel count is initialized using the SPU Channel Count Register in the SPU channel access facility (see page 242).

If SPU interrupts are enabled (SPU_RdMachStat[IE] set to ‘1’), a nonzero SPU Read Event Status Channel count results in an interrupt targeting the SPU.

Access Type Read-blocking

Channel Number x‘0’



Bits	Field Name	Description
0:18	Reserved	Reserved.
Note: For more information about these events, see *Section 9.12 SPU Event Definitions* on page 163.		

1. The SPU Read Event Status Channel has a depth of one. Therefore, its count can be only 0 or 1. The count is set to 1 by privileged software when restoring an SPE context with enabled, pending events.

Bits	Field Name	Description
19	Ms	Multisource synchronization event This event is triggered when the multisource synchronization request has completed. The multi-source synchronization request completes when all pending transfers before a write of the MFC Write Multisource Synchronization Request Channel (see page 149) have completed. This event is triggered immediately if no transfers are pending at the time of the MFC Write Multisource Synchronization Request Channel write. 0 Event has not occurred. 1 Event has occurred and has not been acknowledged.
20	A	Privileged attention event This event is triggered by setting the SPU Attention-Event Request bit in the SPU Privileged Control Register. Access to this register should be limited to privileged software. 0 Event has not occurred. 1 Event has occurred and has not been acknowledged.
21	Lr	Lock line reservation lost event This event is triggered 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. It is not set due to a reservation reset by a local action. 0 Event has not occurred. 1 Event has occurred and has not been acknowledged. This event is set when a snoop external to the MFC causes a lock-line reservation to be reset. The event must *not* be set if the reservation is lost due to a local action.
22	S1	SPU signal notification 1 available event This event is triggered when a processor or a device writes to the SPU Signal Notification 1 Register of the corresponding SPU. 0 Event has not occurred. 1 Event has occurred and has not been acknowledged.
23	S2	SPU signal notification 2 available event This event is triggered when a processor or a device writes to the SPU Signal Notification 2 Register of the corresponding SPU. 0 Event has not occurred. 1 Event has occurred and has not been acknowledged.
24	Le	SPU outbound mailbox available event This event is triggered when the SPU Write Outbound Mailbox Channel count becomes greater than or equal to an implementation-dependent mailbox queue-empty threshold. 0 Event has not occurred. 1 Event has occurred and has not been acknowledged.
25	Me	SPU outbound interrupt mailbox available event This event is triggered when the SPU Write Outbound Interrupt Mailbox Channel count becomes greater than or equal to an implementation-dependent interrupt-mailbox queue-empty threshold. 0 Event has not occurred. 1 Event has occurred and has not been acknowledged.
26	Tm	SPU decrementer event This event is triggered by the transition of the most-significant bit of the SPU decrementer count from ‘0’ to ‘1’. 0 Event has not occurred. 1 Event has occurred and has not been acknowledged.
27	Mb	SPU inbound mailbox available event This event is triggered when the SPU Read Inbound Mailbox Channel count transitions from 0 to a nonzero value. 0 Event has not occurred. 1 Event has occurred and has not been acknowledged.

Note: For more information about these events, see *Section 9.12 SPU Event Definitions* on page 163.

Bits	Field Name	Description
28	Qv	MFC SPU command queue available event This event is triggered by the transition of the MFC SPU command queue from a full state to a not-full state. 0 Event has not occurred. 1 Event has occurred and has not been acknowledged.
29	Reserved	Reserved
30	Sn	MFC list command stall-and-notify event This event occurs when the MFC encounters one or more list commands with the stall-and-notify flag set in the list elements (see *Section 7.4 List Commands and List Elements* beginning on page 63). When this type of MFC command is encountered, the list element is completed. Further list processing is suspended until the stall is acknowledged by the SPU program. 0 Event has not occurred. 1 Event has occurred and has not been acknowledged.
31	Tg	MFC tag-group status update event The tag status event occurs when MFC Read Tag-Group Status Channel (see page 133) is updated based on the tag status update requested by writing the MFC Write Tag Status Update Request Channel (see page 132). 0 Event has not occurred. 1 Event has occurred and has not been acknowledged.
Note: For more information about these events, see *Section 9.12 SPU Event Definitions* on page 163.		

Programming Notes:

Software can cause phantom events in two instances:

1. If software acknowledges or masks an event after the event has incremented the SPU Read Event Status Channel count, before reading the event status from the SPU Read Event Status Channel. In this case, reading the SPU Read Event Status Channel returns data that indicates that the event is no longer present or is disabled.
2. If software resets the interrupting condition of an enabled event (such as reading from a mailbox) before reading the SPU Read Event Status Channel and before acknowledging the event. In this case, reading the event-status register returns data that indicates that the event is still pending, even though the condition that generated the event is no longer present. In this case, the event must still be acknowledged.

To avoid generating phantom events, events should be handled as follows:

- Read the SPU Read Event Status Channel.
- For all events that are to be processed, acknowledge the events by writing the corresponding bits to the SPU Write Event Acknowledgment Channel (see page 161).
- Process the events (for example, read the mailbox, reset, or stop the timer, or read a signal notification register).

Implementation Note:

Hardware determines events by detecting the appropriate channel counts, decrementer count, or SPU channel access operation as described in the following list. (See *Figure 9-1 Logical Representation of SPU Event Support* on page 151 for an illustration of how the events, channels, and registers interact.)

- The MFC tag-group status update event is set when the count for the MFC Read Tag-Group Status Channel (see page 133) changes from 0 to a nonzero value.
- The MFC list command stall-and-notify event is set when the count for the MFC Read List Stall-and-Notify Tag Status Channel (see page 135) changes from 0 to a nonzero value.
- The MFC SPU command queue available event is set when the count for the queued MFC Command Opcode Register (see page 81) changes from 0 (full) to a nonzero (not full) value.
- The SPU inbound mailbox available event is set when the count for the SPU Read Inbound Mailbox Channel (see page 141) changes from 0 to a nonzero value.
- The SPU decrementer event is set when the most-significant bit of the decrementer count changes from ‘0’ to ‘1’. If a 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.
- The SPU outbound mailbox available event is set when the SPU Write Outbound Mailbox Channel (see page 139) count changes from 0 to a nonzero value.
- The SPU outbound interrupt mailbox available event is set when the SPU Write Outbound Interrupt Mailbox Channel (see page 140) count changes from 0 to a nonzero value.
- The SPU signal notification 2 available event is set when the count for the SPU Signal Notification 2 Channel (see page 144) changes from 0 to a nonzero value.
- The SPU signal notification 1 available event is set when the count for the SPU Signal Notification 1 Channel (see page 143) changes from 0 to a nonzero value.
- The lock line reservation lost event is set when SPU Pending Event Register [LR] is set to ‘1’. The SPU Pending Event Register is an internal register. The format of the SPU Pending Event Register is the same as the SPU Read Event Status Channel (see page 153).
- The privileged attention event is set when the SPU Privileged Control Register (see page 239) is written with the Attention-Event Request bit set to ‘1’.
- The multisource synchronization event is set when MFC Write Multisource Synchronization Request Channel (see page 149) count changes from a value of 0 to 1.

9.11.2 SPU Write Event Mask Channel (SPU_WrEventMask)

The SPU Write Event Mask Channel selects which pending events affect the state of the SPU Read Event Status Channel (see page 153). The contents of this channel are retained until a subsequent channel write or an SPU channel access occurs. The current contents of this channel can be accessed by reading the SPU Read Event Mask Channel (see page 159).

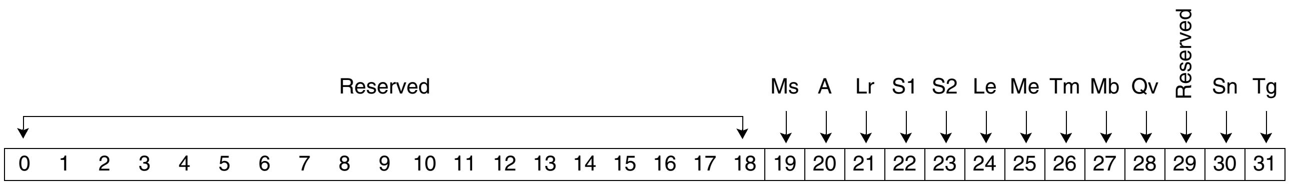
All events are recorded in the SPU Pending Event Register, regardless of the SPU write event mask setting. Events remain pending until cleared by a write channel (**wrch**) instruction to the SPU Write Event Acknowledgment Channel (see page 161) or until privileged software loads the SPU Pending Event Register with a new value using the SPU channel access facility (see page 242). A pending event is cleared even if it is disabled.

Pending events that are disabled and subsequently cleared are not reflected in the SPU Read Event Status Channel (see page 153). Enabling a pending event results in an update of the SPU Read Event Status Channel and an SPU interrupt, if the interrupt is enabled.

This channel is nonblocking and does not have an associated count. A read channel count (**rchcnt**) instruction of this channel always returns 1.

Access Type Write

Channel Number x‘1’



Bits	Field Name	Description
0:18	Reserved	Reserved.
19	Ms	Multisource synchronization event enable. 0 Event is disabled. 1 Event is enabled.
20	A	Privileged attention event enable. 0 Event is disabled. 1 Event is enabled.
21	Lr	Lock line reservation lost event enable. 0 Event is disabled. 1 Event is enabled.
22	S1	SPU signal notification 1 available event enable. 0 Event is disabled. 1 Event is enabled.
23	S2	SPU signal notification 2 available event enable. 0 Event is disabled. 1 Event is enabled.
Note: For more information about these events, see *Section 9.12 SPU Event Definitions* beginning on page 163.		

Bits	Field Name	Description
24	Le	SPU outbound mailbox available event enable. 0 Event is disabled. 1 Event is enabled.
25	Me	SPU outbound interrupt mailbox available event enable. 0 Event is disabled. 1 Event is enabled.
26	Tm	SPU decrementer event enable. Setting this bit to ‘0’ before acknowledging a decrementer event results in the decrementer being stopped, regardless of the decrementer event status. See *Section 9.7* on page 145 for more details 0 Event is disabled. 1 Event is enabled.
27	Mb	SPU inbound mailbox available event enable. 0 Event is disabled. 1 Event is enabled.
28	Qv	MFC SPU command queue available event enable. 0 Event is disabled. 1 Event is enabled.
29	Reserved	Reserved.
30	Sn	MFC list command stall-and-notify event enable. 0 Event is disabled. 1 Event is enabled.
31	Tg	MFC tag-group status update event enable. 0 Event is disabled. 1 Event is enabled.
Note: For more information about these events, see *Section 9.12 SPU Event Definitions* beginning on page 163.		

Implementation Note:

The SPU decrementer must stop if the SPU decrementer event (bit 26) is disabled when an SPU decrementer event is acknowledged. The SPU decrementer event is acknowledged by writing a ‘1’ to bit 26 of the SPU Write Event Acknowledgment Channel (see page 161). The decrementer must stop regardless of the status of the SPU decrementer event at the time of the acknowledgment. The SPU decrementer must not stop if the SPU decrementer event is enabled when the event is acknowledged. Once the SPU decrementer is stopped, no further SPU decrementer events can occur.

9.11.3 SPU Read Event Mask Channel (SPU_RdEventMask)

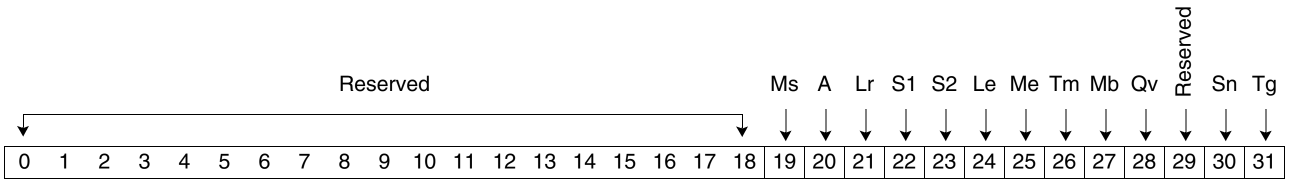
The SPU Read Event Mask Channel is used to read the current value of the event status mask. Reading this channel always returns the last data written by the SPU Write Event Mask Channel (see page 157).

The SPU Read Event Mask Channel allows software to read the state of the event status mask. This channel can be used to avoid software shadow copies of the event status mask and for SPE context save and restore operations.

This channel is nonblocking and does not have an associated count. Whenever a read channel count (**rchcnt**) instruction targets this channel, the count is always returned as 1.

Access Type Read

Channel Number x‘B’



Bits	Field Name	Description
0:18	Reserved	Reserved.
19	Ms	Multisource synchronization event enable. 0 Event is disabled. 1 Event is enabled.
20	A	Privileged attention event enable. 0 Event is disabled. 1 Event is enabled.
21	Lr	Lock line reservation lost event enable. 0 Event is disabled. 1 Event is enabled.
22	S1	SPU signal notification 1 available event enable. 0 Event is disabled. 1 Event is enabled.
23	S2	SPU signal notification 2 available event enable. 0 Event is disabled. 1 Event is enabled.
24	Le	SPU outbound mailbox available event enable. 0 Event is disabled. 1 Event is enabled.
25	Me	SPU outbound interrupt mailbox available event enable. 0 Event is disabled. 1 Event is enabled.
26	Tm	SPU decrementer event enable. 0 Event is disabled. 1 Event is enabled.

Note: For more information about these events, see *Section 9.12 SPU Event Definitions* beginning on page 163.

Bits	Field Name	Description
27	Mb	SPU inbound mailbox available event enable. 0 Event is disabled. 1 Event is enabled.
28	Qv	MFC SPU command queue available event enable. 0 Event is disabled. 1 Event is enabled.
29	Reserved	Reserved.
30	Sn	MFC list command stall-and-notify event enable. 0 Event is disabled. 1 Event is enabled.
31	Tg	MFC tag-group status update event enable. 0 Event is disabled. 1 Event is enabled.

Note: For more information about these events, see *Section 9.12 SPU Event Definitions* beginning on page 163.

9.11.4 SPU Write Event Acknowledgment Channel (SPU_WrEventAck)

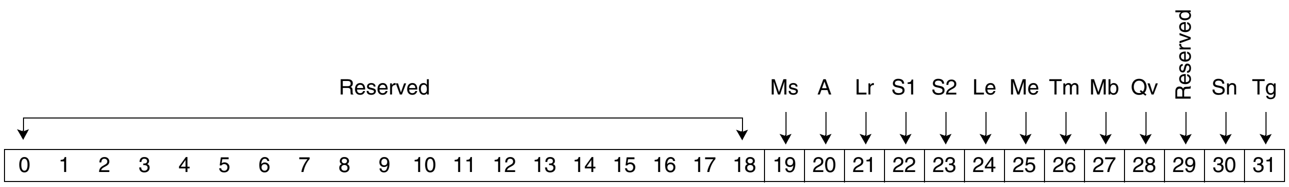
A write to the SPU Write Event Acknowledgment Channel, with specific event bits set, acknowledges that the corresponding events are being serviced by the software. Events that have been acknowledged are reset and resampled. Events that have been reported, but not acknowledged, continue to be reported until acknowledged or until cleared by privileged software using the SPU channel access facility (see page 242).

Disabled events are not reported in the SPU Read Event Status Channel (see page 153). Instead, they are held pending until they are cleared by writing a '1' to the corresponding bit in the SPU Write Event Acknowledgment Channel. Acknowledging a disabled event clears the event, even though it has not been reported. Clearing an event before it occurs can result in a software-induced deadlock. Software should be careful in clearing unreported events. For more on handling these events, see *Section 9.11 SPU Event Facility* beginning on page 150. For more on the events themselves, see *Section 9.12 SPU Event Definitions* beginning on page 163.

This channel is nonblocking and does not have an associated count. Whenever a read channel count (**rchcnt**) instruction targets this channel, the count is always returned as 1.

Access Type Write

Channel Number x‘2’



Bits	Field Name	Description
0:18	Reserved	Reserved.
19	Ms	Multisource synchronization event acknowledgment. 0 Event not acknowledged. 1 Event acknowledged.
20	A	Privileged attention event acknowledgment. 0 Event not acknowledged. 1 Event acknowledged.
21	Lr	Lock line reservation lost event acknowledgment. 0 Event not acknowledged. 1 Event acknowledged.
22	S1	SPU signal notification 1 available event acknowledgment. 0 Event not acknowledged. 1 Event acknowledged.
23	S2	SPU signal notification 2 available event acknowledgment 0 Event not acknowledged. 1 Event acknowledged.
24	Le	SPU outbound mailbox available event acknowledgment 0 Event not acknowledged. 1 Event acknowledged.

Note: For more information about these events, see *Section 9.12 SPU Event Definitions* beginning on page 163.

Bits	Field Name	Description
25	Me	SPU outbound interrupt mailbox available event acknowledgment 0 Event not acknowledged. 1 Event acknowledged.
26	Tm	SPU decrementer event acknowledgment. 0 Event not acknowledged. 1 Event acknowledged.
27	Mb	SPU inbound mailbox available event acknowledgment 0 Event not acknowledged. 1 Event acknowledged.
28	Qv	MFC SPU command queue available event acknowledgment. 0 Event not acknowledged. 1 Event acknowledged.
29	Reserved	Reserved.
30	Sn	MFC list command stall-and-notify event acknowledgment. 0 Event not acknowledged. 1 Event acknowledged.
31	Tg	MFC tag-group status update event acknowledgment. 0 Event not acknowledged. 1 Event acknowledged.

Note: For more information about these events, see *Section 9.12 SPU Event Definitions* beginning on page 163.

Implementation Note:

The SPU decrementer stops if the SPU decrementer event (bit 26) is disabled when an SPU decrementer event is acknowledged. The SPU decrementer event is acknowledged by writing a ‘1’ to bit 26 of the SPU Write Event Acknowledgment Channel. The decrementer stops regardless of the status of the SPU decrementer event at the time of the acknowledgment. The decrementer does not stop if the SPU decrementer event is enabled when the event is acknowledged. Once the decrementer is stopped, no further SPU decrementer events occur. See *Section 9.7.1* beginning on page 145 for more details.

9.12 SPU Event Definitions

The SPU events follow:

- MFC Tag-Group Status Update Event
- MFC List Command Stall-and-Notify Event
- MFC SPU Command Queue Available Event (see page 165)
- SPU Inbound Mailbox Available Event (see page 166)
- SPU Decrementer Event (see page 166)
- SPU Outbound Interrupt Mailbox Available Event (see page 167)
- SPU Outbound Mailbox Available Event (see page 168)
- SPU Signal Notification 2 Available Event (see page 169)
- SPU Signal Notification 1 Available Event (see page 170)
- Lock Line Reservation Lost Event (see page 170)
- Privileged Attention Event (see page 172)
- Multisource Synchronization Event (see page 172)

9.12.1 MFC Tag-Group Status Update Event

The MFC tag-group status update event is used to notify an SPU program that a tag group or groups have completed. It also notifies an SPU program that the MFC Read Tag-Group Status Channel (see page 133) has been updated and can be read without stalling the SPU. See *Section 9.3 MFC Tag-Group Status Channels* beginning on page 126 for more information.

The event occurs when the channel count for the MFC Read Tag-Group Status Channel changes from 0 to 1. When this event occurs, it sets Pend_Event[Tg] to ‘1’. If the event is enabled (that is, SPU_RdEventMask[Tg] is set to ‘1’), the count for the SPU Read Event Status Channel is set to 1 and SPU_RdEventStat[Tg] is set to ‘1’.

The Pend_Event[Tg] bit is set to ‘0’ when a write channel (**wrch**) targets the SPU Write Event Acknowledgment Channel. It is also set to ‘0’ when privileged software updates the SPU Pending Event Register using the SPU channel access facility with the corresponding bit set to ‘0’.

This event must be cleared before issuing any commands for the tag group or groups. For more information about the procedure for using the SPU Tag-Group Status Update event, see *Section 9.3 MFC Tag-Group Status Channels* beginning on page 126.

9.12.2 MFC List Command Stall-and-Notify Event

The MFC list command stall-and-notify event is used to notify an SPU program that a list element within a list command has completed. It also notifies an SPU program that the MFC Read List Stall-and-Notify Tag Status Channel (see page 135) has been updated and can be read without stalling the SPU. See *Section 7.4 List Commands and List Elements* beginning on page 63 for more information.

The event occurs when the channel count for the MFC Read List Stall-and-Notify Tag Status Channel changes from 0 to 1. The count is set to 1 when all the transfers of the list elements with the stall-and-notify flag set, as well as the transfers for all the previous list elements in the list command, are complete with respect to the associated SPE. When this event occurs, it sets Pend_Event[Sn] to ‘1’. If the event is enabled (that is, SPU_RdEventMask[Sn] is set to ‘1’), the count for the SPU Read Event Status Channel is set to 1 and SPU_RdEventStat[Sn] is set to ‘1’.

The Pend_Event[Sn] bit is set to ‘0’ when a write channel (**wrch**) targets the SPU Write Event Acknowledgment Channel with the Sn bit set (that is, SPU_WrEventAck[Sn] is set to ‘1’). It is also set when privileged software updates the SPU Pending Event Register using the SPU channel access facility with the corresponding bit set to ‘0’.

The following procedure handles the MFC list command stall-and-notify event:

1. Issue a read channel (**rdch**) instruction to the SPU Read Event Mask Channel and save the data in the “mask”.[1]
2. Mask the event by issuing a write channel (**wrch**) instruction to the SPU Write Event Mask Channel with SPU_WrEventMask[Sn] set to ‘0’.[1]
3. Acknowledge the event by issuing a write channel (**wrch**) instruction to the SPU Write Event Acknowledgment Channel with SPU_WrEventAck[Sn] set to ‘1’.[1]
4. Issue a read channel (**rdch**) instruction to the MFC Read List Stall-and-Notify Tag Status Channel MFC_RdListStallStat[g_n].
5. Use this information to determine which tag group or tag groups have a list element in the Stall-and-Notify state.
6. Perform the application-specific action with respect to each tag group that has a stalled list element.

   Note: In the following instances, it is essential for the application software to initialize a tag-group specific stall counter to 0 before the list commands are queued for the tag group:

   - When an MFC list contains multiple list elements having the stall-and-notify flag set
   - When a tag group has multiple list commands queued, with elements having the stall-and-notify flag set

   In addition, when multiple 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 list command cannot be changed.
7. Acknowledge and resume each stalled 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 MFC list stall-and-notify handler.

   Note: If application software does not acknowledge all stalled tag groups indicated in the MFC_RdListStallStat[g_n] channel, a second stall-and-notify event does not occur for the unacknowledged tag group.

1. 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. Restore the “mask” by issuing a write channel (**wrch**) instruction to the SPU Write Event Mask Channel with SPU_WrEventMask[Sn] set to ‘1’.[1]
10. Exit the general event handler.[1]

9.12.3 MFC SPU Command Queue Available Event

The MFC SPU command queue available event is used to notify an SPU program that an entry in the MFC SPU command queue is available and that the MFC Command Opcode Channel (see page 117) can be written without stalling the SPU.

The event occurs when the channel count for the MFC Command Opcode Channel changes from a 0 (full) to a nonzero (not full) value. The count is incremented when an MFC DMA command in the MFC SPU command queue is completed. When this event occurs, it sets Pend_Event[Qv] to ‘1’. If the event is enabled (that is, SPU_RdEventMask[Qv] is set to ‘1’), the count for the SPU Read Event Status Channel is set to 1 and SPU_RdEventStat[Qv] is set to ‘1’.

The Pend_Event[Qv] bit is set to ‘0’ when a write channel (**wrch**) targets the SPU Write Event Acknowledgment Channel with the Qv bit set (that is, SPU_WrEventAck[Qv] is set to ‘1’). It is also set to ‘0’ when privileged software updates the SPU Pending Event Register using the SPU channel access facility with the corresponding bit set to ‘0’.

The following procedure handles the MFC SPU command queue available event:

1. Issue a read channel (**rdch**) instruction to the SPU Read Event Mask Channel and save the data in the “mask”.[1]
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’.[1]
3. Acknowledge the event by issuing 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 SPU command queue.
7. If more commands are left to queue, return to step 3.
8. Exit the MFC 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’.[1]
10. Exit the general event handler.[1]

1. 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.

9.12.4 SPU Inbound Mailbox Available Event

The SPU inbound mailbox available event is used to notify an SPU program that a PPE or other device has written to an empty SPU mailbox and that the SPU Read Inbound Mailbox Channel (see page 141) can be read without stalling the SPU.

The event occurs when the channel count for the SPU Read Inbound Mailbox Channel changes from 0 (empty) to a nonzero (not empty) value. When this event occurs, it sets Pend_Event[Mb] to '1'. If this event is enabled (that is, SPU_RdEventMask[Mb] is '1'), the count for the SPU Read Event Status Channel is set to 1 and SPU_RdEventStat[Mb] is set to '1'.

The Pend_Event[Mb] bit is set to '0' when a write channel (**wrch**) targets the SPU Write Event Acknowledgment Channel with the Mb bit set (that is, SPU_WrEventAck[Mb] is set to '1'). It is also set to '0' when privileged software updates the SPU Pending Event Register using the SPU channel access facility with the corresponding bit set to '0'.

The following procedure handles the SPU inbound mailbox available event:

1. Issue a read channel (**rdch**) instruction to the SPU Read Event Mask Channel and save the data in the "mask".[1]
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'.[1]
3. Acknowledge the event by issuing a write channel (**wrch**) instruction to the SPU Write Event Acknowledgment Channel with SPU_WrEventAck[Mb] set to '1'.[1]
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).
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'.[1]
10. Exit the general event handler.[1]

9.12.5 SPU Decrementer Event

The SPU decrementer event is used to notify an SPU program that the decrementer has reached 0. See *Section 9.7.1 SPU Write Decrementer Channel* beginning on page 145 for more information.

The event occurs when the most-significant bit of the decrementer changes from '0' to '1' (negative) value. When this event occurs, it sets Pend_Event[Tm] to '1'. If the event is enabled (that is, SPU_RdEventMask[Tm] is set to '1'), the count for the SPU Read Event Status Channel is set to 1 and SPU_RdEventStat[Tm] is set to '1'.

1. 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 Pend_Event[Tm] bit is set to ‘0’ when a write channel (**wrch**) targets the SPU Write Event Acknowledgment Channel with the Tm bit set (that is, SPU_WrEventAck[Tm] is set to ‘1’). It is also set to ‘0’ when privileged software updates the SPU Pending Event Register using the SPU channel access facility with the corresponding bit set to ‘0’.

The following procedure handles the SPU decrementer event:

1. Issue a read channel (**rdch**) instruction to the SPU Read Event Mask Channel and save the data in the “mask”.[1]
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’.[1]
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 required interval.
5. If a new timer event is required, 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’.[1]
8. Exit the general event handler.[1]

9.12.6 SPU Outbound Interrupt Mailbox Available Event

The SPU outbound interrupt mailbox available event is used to notify an SPU program that a PPE or another device has read from a full SPU Outbound Interrupt Mailbox Register (see page 237) and that the SPU Write Outbound Interrupt Mailbox Channel (see page 140) can be written without stalling the SPU.

The event occurs when the channel count for the SPU Write Outbound Interrupt Mailbox Channel changes from 0 (full) to a nonzero (not full) value. When this event occurs, it sets Pend_Event[Me] to ‘1’. If this event is enabled (that is, SPU_RdEventMask[Me] is set to ‘1’), the count for the SPU Read Event Status Channel is set to 1 and SPU_RdEventStat[Me] is set to ‘1’.

The Pend_Event[Me] bit is set to ‘0’ when a write channel (**wrch**) targets the SPU Write Event Acknowledgment Channel with the Me bit set (that is, SPU_WrEventAck[Me] is set to ‘1’). It is also set to ‘0’ when privileged software updates the SPU Pending Event Register using the SPU channel access facility with the corresponding bit set to ‘0’.

The following procedure handles the SPU outbound interrupt mailbox available event:

1. Issue a read channel (**rdch**) instruction to the SPU Read Event Mask Channel and save the data in the “mask”.[1]
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’.[1]

1. 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 with SPU_WrEventAck[Me] set to a '1'.[1]
4. Obtain the channel count by issuing a read channel count (**rchcnt**) instruction to the SPU Write Outbound Interrupt Mailbox Channel (see page 140).
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 Interrupt Mailbox Channel (see page 140).
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'.[1]
9. Exit the general event handler.[1]

9.12.7 SPU Outbound Mailbox Available Event

The SPU outbound mailbox available event is used to notify an SPU program that either a processor or another device has read from a full SPU Outbound Mailbox Register (see page 102) and that the SPU Write Outbound Mailbox Channel (see page 139) can be written without stalling the SPU.

The event occurs when the channel count for the SPU Write Outbound Mailbox Channel (see page 139) changes from a 0 (full) to a nonzero (not full) value. When this event occurs, it sets Pend_Event[Le] to '1'. If the event is enabled (that is, SPU_RdEventMask[Le] is set to '1'), the count for the SPU Read Event Status Channel is set to 1 and SPU_RdEventStat[Le] is set to '1'.

The Pend_Event[Le] bit is set to '0' when a write channel (**wrch**) targets the SPU Write Event Acknowledgment Channel (see page 161) with the Le bit set to '1' (that is, SPU_WrEventAck[Le] is set to '1'). It is also set to '0' when privileged software updates the SPU Pending Event Register using the SPU channel access facility with the corresponding bit set to '0'.

The following procedure handles the SPU outbound mailbox available event:

1. Issue a read channel (**rdch**) instruction to the SPU Read Event Mask Channel and save the data in the "mask".[1]
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'.[1]
3. Acknowledge the event by issuing a write channel (**wrch**) instruction to the SPU Write Event Acknowledgment Channel with SPU_WrEventAck[Le] set to '1'.[1]
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.

1. 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.

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’.[1]
9. Exit the general event handler.[1]

9.12.8 SPU Signal Notification 2 Available Event

The SPU signal notification 2 available event is used to notify an SPU program that another processor or device has written to an empty SPU Signal Notification 2 Register and that the SPU Signal Notification 2 Channel can be read without stalling the SPU. See *Section 9.6 SPU Signalling Channels* beginning on page 142 for more information.

The event occurs when the channel count for the SPU Signal Notification 2 Channel changes from 0 (empty) to 1 (valid) value. When this event occurs, it sets Pend_Event[S2] to ‘1’. If the event is enabled (that is, SPU_RdEventMask[S2] is set to ‘1’), the count for the SPU Read Event Status Channel is set to 1 and SPU_RdEventStat[S2] is set to ‘1’.

The Pend_Event[S2] bit is set to ‘0’ when a write channel (**wrch**) targets the SPU Write Event Acknowledgment Channel with the S2 bit set (that is, SPU_WrEventAck[S2] is set to ‘1’). It is also set to ‘0’ when privileged software updates the SPU Pending Event Register using the SPU channel access facility with the corresponding bit set to ‘0’.

The following procedure handles the SPU signal notification 2 available event:

1. Issue a read channel (**rdch**) instruction to the SPU Read Event Mask Channel and save the data in the “mask”.[1]
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’.[1]
3. Acknowledge the event by issuing a write channel (**wrch**) instruction to the SPU Write Event Acknowledgment Channel with SPU_WrEventAck[S2] set to a ‘1’.[1]
4. Obtain the channel count by issuing a read channel count (**rchcnt**) instruction to the SPU Signal Notification 2 Channel (see page 144).
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 SPU 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’.[1]
9. Exit the general event handler.[1]

1. 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.

9.12.9 SPU Signal Notification 1 Available Event

The SPU signal notification 1 available event is used to notify an SPU program that another processor or device has written to an empty SPU Signal Notification 1 Register and that the SPU Signal Notification 1 Channel (see page 143) can be read without stalling the SPU. See *Section 9.6 SPU Signalling Channels* beginning on page 142 for more information.

The event occurs when the channel count for the SPU Signal Notification 1 Channel changes from a 0 (empty) to a 1 (valid) value. When this event occurs, it sets Pend_Event[S1] to ‘1’. If the event is enabled (that is, SPU_RdEventMask[S1] is set to ‘1’), the count for the SPU Read Event Status Channel is set to 1 and SPU_RdEventStat[S1] is set to ‘1’.

The Pend_Event[S1] bit is set to ‘0’ when a write channel (**wrch**) targets the SPU Write Event Acknowledgment Channel with the S1 bit set (that is, SPU_WrEventAck[S1] is set to ‘1’). It is also set to ‘0’ when privileged software updates the SPU Pending Event Register using the SPU channel access facility with the corresponding bit set to ‘0’.

The following procedure handles the SPU signal notification 1 available event:

1. Issue a read channel (**rdch**) instruction to the SPU Read Event Mask Channel and save the data in “mask”.[1]
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’.[1]
3. Acknowledge the event by issuing a write channel (**wrch**) instruction to the SPU Write Event Acknowledgment Channel with SPU_WrEventAck[S1] set to ‘1’.[1]
4. Obtain the channel count by issuing a read channel count (**rchcnt**) instruction to the SPU Signal Notification 1 Channel (see page 143).
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 SPU 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’.[1]
9. Exit the general event handler.[1]

9.12.10 Lock Line Reservation Lost Event

The lock line reservation lost event is used to notify an SPU program of a bus action that resulted in the loss of the reservation on an aligned unit of real storage, a reservation granule. An SPU program acquires a reservation by issuing a **getllar** command. The reservation is lost when another processor or device modifies one or more bytes in the reservation granule. The reservation can also be lost if privileged software writes the Flush bit in the MFC Atomic Flush register (MFC_Atomic_Flush[F] is set to ‘1’). See *Section 7.8.1 Get Lock Line and Reserve Command* beginning on page 70 for more information.

1. 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 event occurs when the reservation is lost. When this event occurs, it sets Pend_Event[Lr] to ‘1’. If the event is enabled (that is, SPU_RdEventMask[Lr] is set to ‘1’), the count for the SPU Read Event Status Channel is set to 1, and SPU_RdEventStat[Lr] is set to ‘1’.

The Pend_Event[Lr] bit is set to ‘0’ when a write channel (**wrch**) targets the SPU Write Event Acknowledgment Channel with the Lr bit set (that is, SPU_WrEventAck[Lr] is set to ‘1’). It is also set to ‘0’ when privileged software updates the SPU Pending Event Register using the SPU channel access facility with the corresponding bit set to ‘0’.

Programming Note:

A **getllar** command creates a reservation on a unit of real storage called the reservation granule. The reservation is lost when another processor or device modifies any bytes in the reservation granule. When the reservation is lost, a lock line reservation lost event is presented to the SPU. While waiting for the event to be presented, the SPU can stall.

Some operating systems or hypervisors perform operations such as page stealing, page swapping, and copy on write that change the effective-address-to-real-address mapping. If one of these operations occurs after an SPU acquires a reservation, an update performed by another processor or device might modify a *different* reservation granule. Thus, the reservation held by the SPU is *not* lost, and the expected lock line reservation lost event is *not* presented. Not all operating systems and hypervisors are implemented in a manner that can cause this condition. Contact your system provider to determine if you need to plan for this possibility.

When using the lock line reservation lost event, software should avoid this condition or any condition that might prevent an expected event from being presented to an SPU. Software should also be designed to handle this situation if it does occur. To break out of an indefinite wait, the software design should include some form of a watchdog timer (contact your system provider about the preferred form of the watchdog timer). Software should reissue the **getllar** command if the expected lock line reservation lost event does not occur within the time period specified in the watchdog timer.

The following procedure handles the lock line reservation lost event:

1. Issue a read channel (**rdch**) instruction to the SPU Read Event Mask Channel and save the data in the “mask”.[1]
2. Mask the event by issuing a write channel instruction to the SPU Write Event Mask Channel with SPU_WrEventMask[Lr] set to ‘0’.[1]
3. Acknowledge the event by issuing a write channel (**wrch**) instruction to the SPU Write Event Acknowledgment Channel with SPU_WrEventAck[Lr] set to ‘1’.[1]
4. Perform the application-specific function in response to system modification of data in the lock-line area.

   This is typically 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 acting on that data.
5. Exit the lock line reservation lost event handler.

1. 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.

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’.[1]
7. Exit the general event handler.[1]

9.12.11 Privileged Attention Event

The privileged attention event is used to notify an SPU program that privileged software is requesting attention from an SPU program. Privileged software requests attention by writing ‘1’ to the Attention Event Request bit in the SPU Privileged Control Register (see page 239) (that is, SPU_PrivCntl[A] is set to ‘1’).

The event occurs when SPU_PrivCntl[A] is set to ‘1’. When this event occurs, it sets Pend_Event[A] to ‘1’. If the event is enabled (that is, SPU_RdEventMask[A] is set to ‘1’), the count for the SPU Read Event Status Channel is set to 1 and SPU_RdEventStat[A] is set to ‘1’.

The Pend_Event[A] bit is set to ‘0’ when a write channel (**wrch**) targets the SPU Write Event Acknowledgment Channel with the A bit set (that is, SPU_WrEventAck[A] is set to ‘1’). It is also set to ‘0’ when privileged software updates the SPU Pending Event Register using the SPU channel access facility with the corresponding bit set to ‘0’.

The following procedure handles the privileged attention event:

1. Issue a read channel (**rdch**) instruction to the SPU Read Event Mask Channel and save the data in the “mask”.[1]
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’.[1]
3. Acknowledge the event by issuing a write channel (**wrch**) instruction to the SPU Write Event Acknowledgment Channel with SPU_WrEventAck[A] set to ‘1’.[1]
4. Perform the application-specific function in response to a privileged attention event.

   This signals 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’.[1]
7. Exit the general event handler.[1]

9.12.12 Multisource Synchronization Event

A multisource synchronization event is used to notify an SPU program that a multisource synchronization request has completed. Multisource synchronization is requested by writing to the MFC Write Multisource Synchronization Request Channel (see page 149). Data written to this channel is ignored. However, since all the bits are reserved, software should write a value of 0 for compatibility with future enhancements.

1. 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 event occurs when the channel count for the MFC Write Multisource Synchronization Request Channel changes from 0 to 1. When this event occurs, it sets Pend_Event[Ms] to '1'. If the event is enabled (that is, SPU_RdEventMask[Ms] is set to '1'), the count for the SPU Read Event Status Channel is set to 1 and SPU_RdEventStat[Ms] is set to '1'.

The Pend_Event[Ms] bit is set to '0' when a write channel (**wrch**) targets the SPU Write Event Acknowledgment Channel with the Ms bit set (that is, SPU_WrEventAck[Ms] is set to '1'). It is also set to '0' when privileged software updates the SPU Pending Event Register using the SPU channel access facility with the corresponding bit set to '0'.

The multisource synchronization event must be cleared before issuing the multisource synchronization request.

The following procedure handles the multisource synchronization event:

1. Issue a read channel (**rdch**) instruction to the SPU Read Event Mask Channel and save the data in the "mask".[1]
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'.[1]
3. Acknowledge the event by issuing a write channel (**wrch**) instruction to the SPU Write Event Acknowledgment Channel with SPU_WrEventAck[Ms] set to '1'.[1]
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'.[1]
7. Exit the general event handler.[1]

1. 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.

10. Storage Access Ordering

As shown in *Figure 10-1* on page 176, there are multiple storage domains in the Cell Broadband Engine Architecture (CBEA). The *PowerPC Architecture, Book II* defines storage access ordering and synchronization facilities for the main storage domain. The *Synergistic Processor Unit Instruction Set Architecture* document defines storage access ordering and synchronization facilities for the local storage domain and channel domain.

In a CBEA-compliant processor, there can be multiple local storage and channel domains but only one main storage domain. The ordering of accesses between these different domains is described in the following sections. This description assumes that the local storage is accessed from the main storage domain with the storage attribute of caching inhibited. The CBEA also defines direct memory access (DMA) facilities in the memory flow controller (MFC) for initiating storage accesses in both domains and additional synchronization facilities for main storage beyond those described in the *PowerPC Architecture, Book II*. For more information, also see *Section 6 Memory Flow Controller* beginning on page 51, *Section 7 MFC Commands* beginning on page 55, *Section 8 Problem-State Memory-Mapped Registers* beginning on page 79, and *Section 9 Synergistic Processor Unit Channels* beginning on page 113.

A synergistic processor unit (SPU) initiates accesses within the local storage and channel domains. A Power Processor Element (PPE) initiates accesses within the main storage domain. The MFC can initiate accesses in both the main storage and local storage domains. Local storage can have an alias in the main storage. Therefore, it is shown both in the local storage domain and in the main storage domain. Access to local storage using the alias adheres to the ordering rules for the main storage domain.

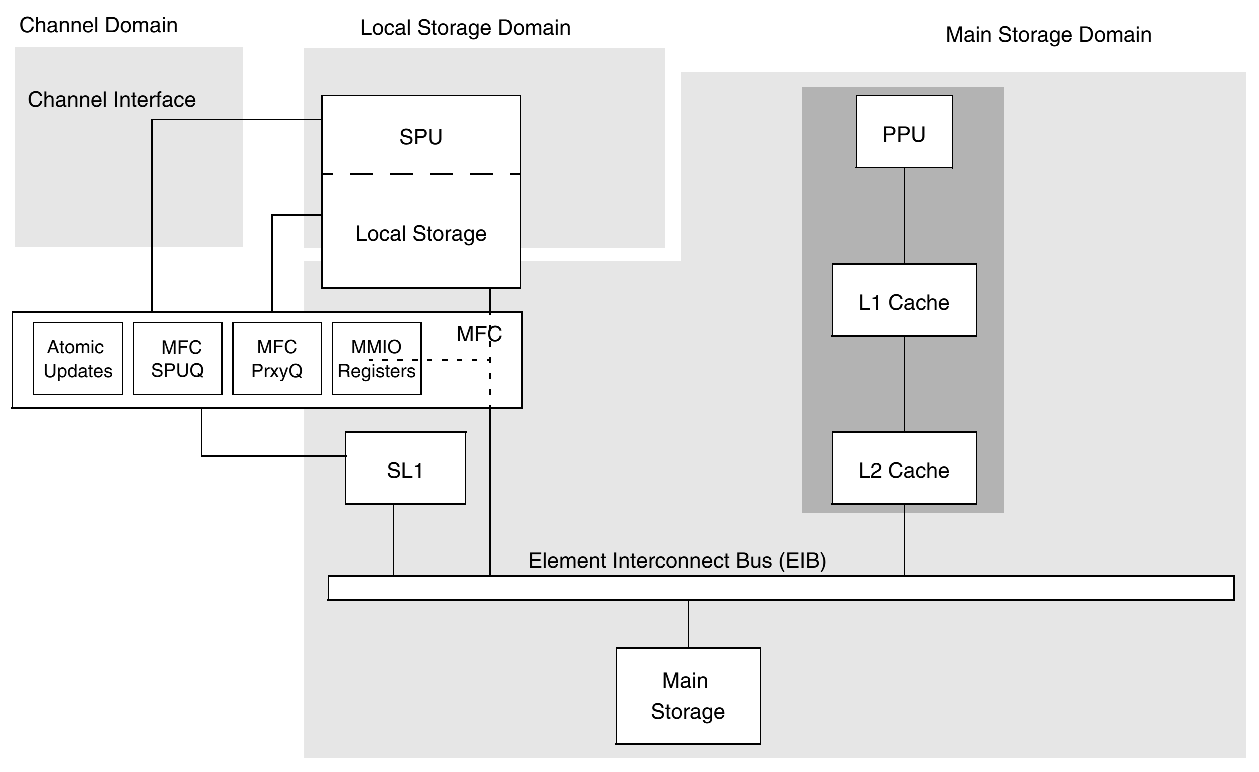
Examples of the access ordering are given in *Appendix F Examples of Access Ordering* beginning on page 321.

As shown in *Figure 10-1* on page 176, the MFC maintains separate command queues for MFC SPU commands and MFC proxy commands. MFC synchronization commands that are placed in the MFC proxy queue only order the MFC commands within the MFC proxy command queue. MFC synchronization commands issued by the SPU only order MFC commands within the MFC SPU command queue.

When an MFC synchronization command finishes processing, both the local storage access and the main storage access are performed. Therefore, subsequent accesses are ordered whether they are initiated by the SPU directly by loads or stores to local storage or are initiated indirectly by channel commands.

When the status returned by a read channel instruction that targets the MFC Read Tag-Group Status Channel (see page 133) indicates that a **put** command is complete, the local storage accesses are complete. The accesses are ordered with respect to the SPU. However, the main storage accesses might not be complete. The accesses are not ordered with respect to other processors and devices. For a **get** command, both the local storage and main storage accesses are complete and ordered with respect to other processors and devices.

Figure 10-1. Storage Domains in a CBEA-Compliant Processor



MFC: Memory flow controller
MFC SPUQ: Memory flow controller synergistic processor unit command queue
MFC PrxyQ: Memory flow controller proxy command queue
MMIO Registers: Memory-mapped input-output registers
PPU: PowerPC processor unit
SL1: First-level cache for DMA transfers between local storage and system memory
SPU: Synergistic processor unit

The **putqlluc** MFC atomic command is placed into the MFC SPU command queue along with other commands. Because this command is queued, it executes independent of any pending immediate **getllar**, **putllc**, and **putlluc** MFC atomic update commands.

The **putqlluc** command creates a tag-specific **fence** even though there is no **<f>** modifier. Therefore, it uses the MFC tag parameter. To determine when the **putqlluc** command is complete, software must wait for tag-group completion. When completed, all accesses from earlier-issued commands with the same tag ID in the MFC SPU command queue and the accesses for the **putqlluc** command are completed. Due to the tag-specific **fence** created by the **putqlluc** command, the local storage and main storage accesses performed by the **putqlluc** are ordered with respect to all earlier-issued commands with the same tag (tag group) in the MFC SPU command queue.

10.1 Order of Command Execution

Both *PowerPC Architecture, Book II* and the *Synergistic Processor Unit Instruction Set Architecture* document have a sequential execution model. In this model, from a software viewpoint, instructions appear to be executed in the order specified by the program, or *program order*. However, the order in which storage accesses are performed can be different from the program order.

The following terms are used in the description of storage access ordering:

Term	Definition
Main storage	The effective-address space. It consists physically of real memory (whatever is external to the memory-interface controller), SPU LSs, memory-mapped registers and arrays, memory-mapped I/O devices, and pages of virtual memory that reside on disk. It does not include caches or execution-unit register files.
MFC effective address access	Accesses to main storage using the effective address in the main storage domain.
MFC LSA access	Accesses to local storage using the local storage address (LSA) in the local storage domain.
MFC put commands	The collection of all MFC commands that transfer data from local storage to the main storage domain. See *Section 7.6 Put Commands (Local Storage to Main Storage)* beginning on page 65.
MFC get commands	The collection of all MFC commands that transfer data from the main storage domain to local storage. See *Section 7.5 Get Commands (Main Storage to Local Storage)* beginning on page 64.

10.2 Main Storage Domain Access Ordering

The storage model for the CBEA is weakly consistent. This model incorporates the same weakly consistent model that the PowerPC Architecture supports for a PPE. This model provides an opportunity for improved performance over a model that has stronger consistency rules. However, it places the responsibility on the programmer or programming tools to ensure that ordering or synchronization instructions, commands, or command modifiers are properly placed when the storage is shared by multiple units in the CBEA. *PowerPC Architecture, Book II* defines PPE storage access ordering and instructions. For DMA operations initiated by the memory flow controllers found in the Synergistic Processor Elements (SPEs), the storage ordering command modifiers, fence and barrier, are provided as are the synchronization commands **mfcsync**, **mfceieio**, and **barrier**. For more information about these facilities, see *Section 7.9 MFC Synchronization Commands* beginning on page 73.

The following rules apply to main storage access order in the main storage domain.

- If two PowerPC processor unit (PPU) store instructions specify storage locations that are both caching inhibited and guarded, the corresponding storage accesses are performed in program order with respect to the main storage accesses of any PPEs, SPEs, or devices.
- All accesses resulting from a single MFC **put** command specifying storage that is both caching inhibited and guarded are performed in sequential (increasing) address order in the main storage domain.
- All accesses resulting from a single **put** list command specifying storage that is both caching inhibited and guarded are performed in sequential (increasing) address order for each list element and in list order between elements with respect to the main storage accesses of any PPEs, SPEs, or devices.

- A PPU load instruction might depend on the value returned by a preceding load instruction because the value is used to compute the effective address specified by the second load. In that case, the corresponding storage accesses are performed in program order with respect to any PPEs, SPEs, or devices to the extent required by the associated memory-coherence-required attributes.[1] This applies even if the dependency has no effect on program logic (for example, the value returned by the first load is ANDed with zero and then added to the effective address specified by the second load).
- An MFC atomic command might be followed by a read channel (**rdch**) from the MFC Read Atomic Command Status Channel that returns a status indicating completion of the command. In this case, the MFC atomic update access to main storage is performed before any main storage accesses specified by an MFC command that was issued later by the SPU through the channel interface.
- When an MFC command queue process is suspended by a PPE memory-mapped I/O (MMIO) operation, all earlier main storage accesses are performed with respect to the PPE that caused the suspension before any later MFC main storage accesses are performed when the MFC command queue process is no longer suspended. The MFC multisource synchronization facility (see page 108) defines a facility that can be used when it is necessary to ensure that all earlier MFC main storage accesses are performed with respect to any and all PPU and MFC main storage accesses.
- When the PPU executes a **sync** or **eieio** instruction, a memory barrier is created that orders applicable storage accesses. When an MFC issues an **mfcsync** or **mfceieio** command, a memory barrier is created that orders applicable main storage accesses in pairs as described below.

  For example, let A be a set of main storage accesses that includes all main storage accesses associated with instructions or applicable MFC commands preceding the barrier-creating instruction or command. Let B be a set of main storage accesses that includes all main storage accesses associated with instructions or applicable MFC commands following the barrier-creating instruction or command. For each applicable pair a_i,b_j of main storage accesses such that a_i is in A and b_j is in B, the memory barrier ensures that a_i will be performed with respect to any PPEs, SPEs, or devices to the extent required by the associated memory-coherence-required attributes,[1] before b_j is performed with respect to that PPE, SPE, or device.

  The memory barrier orders main storage accesses performed by one unit. Some memory barriers have an additional cumulative property, which orders certain main storage accesses done by other units.

  For example, let P1 represent the initiator of main storage accesses and a memory barrier. The ordering done by the memory barrier is said to be "cumulative," if it also orders main storage accesses that are performed by a PPE, or an SPE, or a device other than P1, as follows:

  - A includes all applicable main storage accesses by any PPE, SPE, or other device that were performed with respect to P1 before the memory barrier was created.
  - B includes all applicable storage accesses by any such PPE, SPE, or device that are performed *after* a load instruction or **get** command executed by that PPE, SPE, or device has returned the value stored by a store instruction or **put** command that is performed after the barrier-creating instruction or command (that is, a store instruction or **put** command that is in B).

  Note: The recursion in the definition of B is intentional. Initially B contains the storage accesses performed by P1 after the memory barrier. Repetitively applying the definition for B, the set of storage accesses in B increases with each repetition.

1. The phrase "to the extent required by the associated memory-coherence-required attributes" refers to the memory-coherence-required attribute, if any, associated with each main storage access. This phrase does not apply to storage accesses in the local storage domain.

For storage accesses to storage that does not have the caching inhibited and guarded attribute, no ordering should be assumed for storage accesses caused by a single instruction (that is, by an instruction for which the access is not atomic) or by a single MFC command (that is, by an MFC command for which the access is not atomic). No means are provided for controlling the order.

10.3 Local Storage Domain Access Ordering

The following rules apply to local storage access order in the local storage domain. The access order for SPU channel reads and writes is also described. The MFC DMA commands perform a local storage access using the local storage address (LSA) parameter, which is called an MFC LSA access.

- When an SPU executes either a **sync** or a **dsync** instruction, a memory barrier is created that arranges the SPU instructions in the following order:
  - Earlier load instructions
  - Earlier store instructions
  - Earlier channel read instructions that are issued before later-issued channel read instructions
  - Channel write instructions
  - Later load instructions
  - Later store instructions
- The MFC **mfcsync** and **mfceieio** commands order all earlier-issued commands in the same queue and with the same tag relative to all later-issued commands in the same command queue with the same tag because they create a tag-specific barrier even though there is no **<b>** modifier. MFC LSA accesses for all earlier-issued commands are performed before MFC LSA accesses for any later-issued commands; that is, those following the **mfcsync** and **mfceieio** commands.
- An MFC **barrier** command orders all earlier-issued commands in the same queue relative to all later-issued commands in the same queue. MFC LSA accesses for all earlier-issued commands are performed before MFC LSA accesses for any later-issued commands; that is, those following the MFC **barrier** command.
- An MFC **put** or **get** command with a tag-specific **fence** orders all earlier-issued commands with the same tag and in the same queue relative to this command. MFC LSA accesses for all earlier-issued commands in the same queue and with the same tag are performed before any MFC LSA accesses for an MFC **put** or **get** command with a tag-specific **fence**.
- An MFC **put** or **get** command with a tag-specific **barrier** orders all earlier-issued commands in the same queue relative to all later-issued commands with the same tag and in the same queue. MFC LSA accesses for all earlier-issued commands in the same queue and with the same tag are performed before any MFC LSA accesses for later-issued commands with the same tag, including the command with the tag-specific **barrier**.
- The MFC LSA access is complete for a queued **put** or **get** command when a channel read instruction that targets the MFC Read Tag-Group Status Channel (see page 133) returns a status that indicates the tag group associated with the **put** or **get** command is complete.
- An immediate MFC atomic command that is followed by a read from the MFC Read Atomic Command Status Channel (see page 137) returns status indicating completion of the command. In this case, the MFC atomic update access involving local storage is completed before any later-issued SPU load or store instruction.
- All earlier MFC LSA accesses are performed when the MFC command queue process is suspended by a PPE MMIO operation before any later MFC LSA accesses are performed once the MFC command queue process is resumed.

10.4 Cross-Domain Storage Access Order

Cross-domain storage access order specifies that the results of a strongly ordered access sequence in the main storage domain targeting a single SPE will be seen in the same relative order when accessed by an SPU with a strongly ordered sequence in the local storage domain of that SPE. It also specifies that the results of a strongly ordered access sequence in the local storage domain of a specific SPE will be seen in the same relative order in the main storage domain access directed at that SPE. The SPU Inbound Mailbox Register (see page 103), the SPU Signal Notification 1 Register (see page 106), and the SPU Signal Notification 2 Register (see page 107) are the only MMIO registers for which cross-domain storage access order applies.

Cross-domain storage access order is critical when a main storage domain update of SPE local storage is performed and an SPE is informed using an SPE communication MMIO register.

For examples of how to use the cross-domain storage access order, see examples 15 through 18 in *Appendix F* on page 321.

10.5 Cumulative Access Ordering

The MFC Multisource Synchronization Register (see page 109) defines a facility to achieve cumulative ordering across the local storage and main storage domains.

10.6 MFC Overlapped Accesses

An MFC access to an effective address range that maps to its own local storage (the local storage alias) can produce unpredictable results when the translated effective address area overlaps the local storage area. (For more information, see *Section 3.2.1.2 Local Storage Access Exceptions* on page 42.)

When valid results occur, the access specified by the local storage address (LSA) is still considered to be performed in the local storage domain. The local storage access specified by the effective address is still considered to be performed in the main storage domain. Ordering rules for each domain still apply.

10.7 Atomic Accesses

An access is single-copy atomic, or just atomic, if it is always performed in its entirety with no visible fragmentation. Atomic accesses are therefore serialized; each happens in its entirety in some order, even when that order is not specified in the program, or enforced between processors. All SPU loads and stores to local storage are atomic.

Single-copy atomicity (see page 42) describes atomicity in the main storage domain. All MFC and PPU accesses to local storage, which are defined as atomic in single-copy atomicity, are also defined as atomic in the local storage domain. Therefore, SPU, MFC, and PPU atomic accesses to local storage are performed in their entirety. No fragmentation of these accesses is observed by any other SPU, MFC, or PPU.

10.8 Store Combining

If two PowerPC store instructions specify storage locations that are both caching inhibited and not guarded, such stores can be combined into one access by a CBEA-compliant processor. If the two store instructions were each single-copy atomic and were combined by a CBEA-compliant processor, the combined store is not required to be single-copy atomic. However, the original single-copy atomicity of the original stores must be preserved. If a **sync** or **eieio** instruction separates the store instructions, combining cannot occur.

10.9 Storage Ordering of I/O Accesses

A *coherence domain* consists of all processors, MFCs, and devices with access to main storage and all interfaces to main storage. Mechanisms outside the coherence domain can initiate memory reads and writes. The PowerPC Architecture requires these accesses to be performed within the coherence domain in the order in which they enter the coherence domain. They must also be performed as coherent accesses. This is called a strongly ordered I/O access.

The CBEA requires an implementation to support strongly ordered I/O accesses to be compatible with the PowerPC Architecture. The CBEA also allows an implementation to support a weakly ordered I/O mode. A weakly ordered mode can provide for better efficiency of I/O accesses in certain cases. However, if the weakly ordered mode is provided, the implementation must also provide software with a means of ordering storage accesses when strong ordering is required.

11. SPU Isolation Facility

The Cell Broadband Engine Architecture (CBEA) includes an optional facility that enables privileged software and applications to isolate and load a code image into one or more of the synergistic processor units (SPUs). The SPU isolation facility ensures that the code image loaded into the associated local storage has not been altered by any means.

This section describes only those aspects of the isolation facility that relate to the CBEA. Many details are implementation dependent. For more specific implementation information, contact the system or processor manufacturer.

The SPU isolation facility consists of bits within the registers and channels mentioned in this section and includes:

- SPU Run Control Register (see page 96)
- SPU Status Register (see page 97)
- SPU Read Machine Status Channel (see page 147)
- SPU Privileged Control Register (see page 239)

11.1 SPU Isolation Facility Features

The SPU isolation facility consists of six principal architectural features:

- An SPU isolated execution environment, independent for each SPU in a CBEA-compliant processor.
- An exit function and a load function for moving between the SPU nonisolated execution environment and the SPU isolated execution environment
- An authentication and decryption master key
- A validation procedure for ensuring that the code image loaded when entering the SPU isolated execution environment has not been altered by any means
- Support for random number generation
- A persistent storage that retains its state between isolation sessions

The SPU isolated execution environment allows an application loaded into an isolated area of local storage to execute without being modified, observed, or compromised by any ordinary means external to that SPU. (The isolated area is discussed in the next paragraph.) In this environment, the SPU Next Program Counter does not control where the SPU starts executing instructions. The initial instruction executed when entering the SPU isolated execution environment is implementation dependent.

When initiating a transition into an SPU isolated execution environment and while operating in this environment, an area of local storage starting at address zero is isolated from the system (this area is called the isolated local storage area or the isolated area). Only the associated SPU can access the isolated area. Access from all other processors and devices in the system must not be allowed. The size of the isolated local storage area is implementation dependent. The remaining area, or open area, of local storage is not isolated and remains accessible. The open area can be used for data transfers, control, and communications between the rest of the system and the SPU operating in the SPU isolated execution environment. In addition, internal and external accesses to all debug, test, performance monitoring, and diagnostic interfaces for the associated SPU must be disabled.

The SPU isolation facility provides two transition functions, exit and load, to change the SPU between the two code execution environments: nonisolated and isolated. An exit function must be initiated to change an SPU from an SPU isolated execution environment to an SPU nonisolated execution environment. The exit function erases all SPU states before exiting from the SPU isolated execution environment. A load function must be initiated to change an SPU from an SPU nonisolated execution environment to an SPU isolated execution environment. The load function loads a code image into the local storage and begins executing the image in the SPU isolated execution environment. A CBEA-compliant processor can implement just the exit function, or both the exit and load functions. The load function cannot be implemented without an exit function.

The SPU isolation facility requires a CBEA-compliant processor with nonvolatile storage for holding an authentication and decryption master key. The size of the nonvolatile storage is implementation dependent. The nonvolatile storage must not be accessible by any software, processor, or device in the system. The authentication and decryption master key is intended for exclusive use of the validation procedure.

As part of the load function, a validation procedure is performed when entering the SPU isolated execution environment. This procedure must ensure that no changes have been made to the code image by altering the external source image that supplies the code, by interfering with the loading operation, or by any other means. The validation procedure should use the authentication and decryption master key as part of the validation process. The validation procedure is implementation dependent. For more information, contact the system or processor manufacturer.

The SPU isolation facility also requires support for a random number generation (RNG) function. The details of the RNG function are implementation dependent. An implementation can either allow access to the RNG function in any operating environment or restrict access to it from any environment.

The exit function erases all information from the previous application. Therefore, the SPU isolation facility requires a persistent storage that retains its value even after exiting the SPU isolated execution environment. The persistent storage is not required to be nonvolatile. The persistent storage enables an application to transfer data from one isolated session to another. Access to the persistent storage must only be allowed by an SPU operating in the SPU isolated environment. After a successful load function, read and write access should be allowed to the persistent storage. An implementation must provide a method for an SPU application to change the access restrictions of the persistent storage after entering the SPU isolated execution environment (for example, by removing the read and write accesses). Once the access restrictions are changed, a new load function must be initiated to restore the read and write accesses. The size of the persistent storage area and the method used to access the persistent storage are implementation dependent.

Note: The persistent storage is typically associated with a physical SPU. Therefore, software must provide a method either to ensure that the physical SPU is not changed between two isolation sessions for a given application, or to virtualize the persistent storage. Furthermore, all due caution should be taken by software before loading any code not validated with the authentication and decryption master key to prevent malicious tampering with the persistent storage.

11.2 SPU Operating States

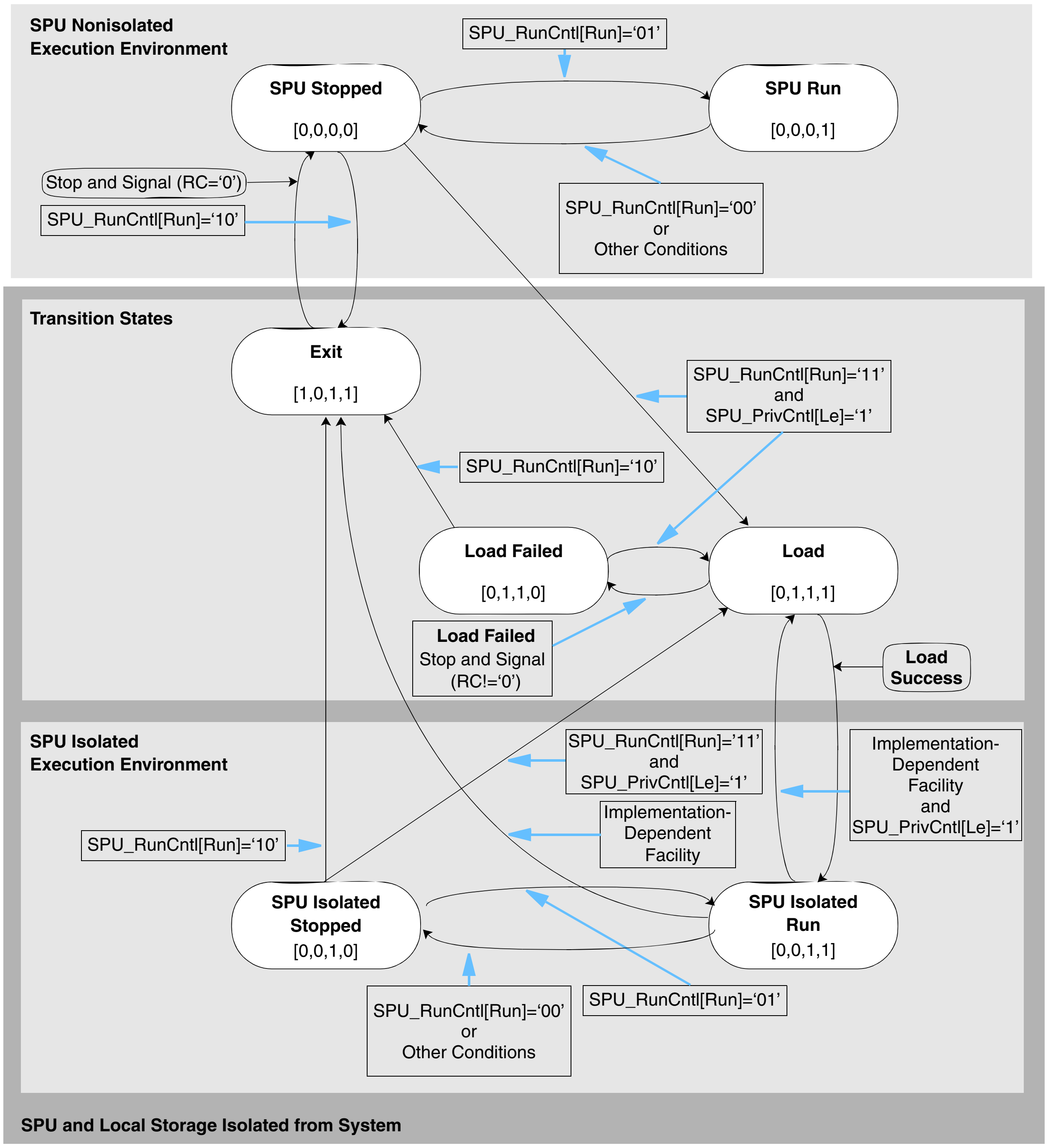
Support for the SPU isolation facility is optional for a CBEA-compliant processor. In addition, an implementation can implement a subset of the SPU isolation facility features, specifically the exit function. An SPU that implements the full isolation facility has seven states of operation compared to two states for an implementation that does not support the isolation facility. If only the exit function is implemented, there are three operational states. The following seven states are implemented with the optional SPU isolation facility:

- SPU Stopped—Stopped in a nonisolated state (all SPU implementations)
- SPU Run—Running in a nonisolated state (all SPU implementations)
- Exit—Performing an isolated exit function (SPU with full or exit isolation support)
- Load—Performing an isolated load function (SPU with full isolation support only)
- Load Failed—The load function failed (SPU with full isolation support only)
- SPU Isolated Run—SPU running in an isolated state (SPU with full isolation support only)
- SPU Isolated Stopped—SPU stopped in an isolated state (SPU with full isolation support only)

Figure 11-1 on page 186 illustrates the SPU states and the methods for transitioning between the states. The three lightly shaded boxes represent the two SPU operating environments (isolated and nonisolated) and the transition between these two environments. When the SPU is in any isolated or transition state, the SPU and the isolated area of local storage cannot be accessed from any other processing or data transfer resource within the system. This is shown as the darker shaded box in the figure.

Figure 11-1 also indicates how the setting of several bits in the SPU Status Register identifies the current SPU state. An application can control the transition between states by writing the SPU Run Control Register or an implementation-dependent facility that is only accessible to an SPU operating in the SPU isolated execution environment. For simplicity, only the values written to the SPU Run Control Register that cause a state transition are shown. All other values written to the SPU Run Control Register remain in their current state.

Figure 11-1. SPU State Transitions


SPU Nonisolated Execution Environment
SPU_RunCntl[Run]='01'
SPU Stopped [0,0,0,0]
SPU Run [0,0,0,1]
Stop and Signal (RC='0')
SPU_RunCntl[Run]='10'
SPU_RunCntl[Run]='00' or Other Conditions
Transition States
Exit [1,0,1,1]
SPU_RunCntl[Run]='11' and SPU_PrivCntl[Le]='1'
SPU_RunCntl[Run]='10'
Load Failed [0,1,1,0]
Load [0,1,1,1]
Load Failed Stop and Signal (RC!='0')
Load Success
SPU Isolated Execution Environment
SPU_RunCntl[Run]='11' and SPU_PrivCntl[Le]='1'
Implementation-Dependent Facility and SPU_PrivCntl[Le]='1'
Implementation-Dependent Facility
SPU_RunCntl[Run]='10'
SPU Isolated Stopped [0,0,1,0]
SPU Isolated Run [0,0,1,1]
SPU_RunCntl[Run]='00' or Other Conditions
SPU_RunCntl[Run]='01'
SPU and Local Storage Isolated from System

Notes:

- The numbers in brackets after the state names indicate how the following bits are set in SPU_Status: E (SPU isolation exit status), L (SPU isolation load status), IS (SPU isolated state), and R (SPU run status).
- SPU_RdMachStat[IS] (isolation status) is set to the same value as SPU_Status[IS] (SPU isolated state).

Privileged Mode Environment

Section 12 through *Section 23* describe the instructions and facilities that are provided by the Cell Broadband Engine Architecture (CBEA) for operating environment software, such as an operating system or a hypervisor. The facilities in the rest of this document are only available to applications running in a privileged mode, either privilege 1 mode or privilege 2 mode. Collectively, these facilities are called the privileged mode environment.

12. Overview

Each of the privileged facilities with their related registers is discussed in the following sections. *Table 12-1 SPE Privilege 1 Memory Map* on page 190, *Table 12-2 SPE Privilege 2 Memory Map* on page 193, and *Table 12-3 PPE Privilege 1 Memory Map* on page 195 list the privileged registers.

- *Section 13 PowerPC Architecture, Book III Compatibility* beginning on page 197
- *Section 14 Storage Addressing* beginning on page 199
- *Section 14.4 Real-Mode Storage Control Facilities* beginning on page 217
- *Section 15 MFC Privileged Facilities* beginning on page 221
- *Section 16 SPU Privileged Facilities* beginning on page 239
- *Section 17 SPE Context Save and Restore* beginning on page 247
- *Section 18 PPE Address Range Facility* beginning on page 249
- *Section 19 Cache Replacement Management Facility* beginning on page 255
- *Section 20 Resource Allocation Management* beginning on page 259
- *Section 21 Interrupt Facilities* beginning on page 261
- *Section 22 Power Management* beginning on page 285
- *Section 23 Version Control* beginning on page 287

12.1 Privileged Mode Facility Organization

The facilities described in the privileged mode environment are classified as either privilege 1 or privilege 2. These designations relate to a suggested hierarchy of privileged access. The access hierarchy is defined to support a 2-level operating environment. An example of such an operating environment is when multiple operating systems run concurrently on top of a more privileged hypervisor. This type of operating environment implements logical partitioning.

Privilege 1 registers are the most privileged. They are intended to be accessed by a hypervisor or by firmware operating in hypervisor mode (HV = ‘1’ and PR = ‘0’), typically when supporting logical partitioning. (For information about hypervisor mode, see *PowerPC Architecture, Books I-III*.) Privilege 2 registers are intended for privileged operating system code running in the HV = ‘0’ and PR = ‘0’ mode. When a single-level operating environment exists, firmware and the privileged operating system typically combine privilege 1 and privilege 2 resources into one privileged level.

The page table should be set up so that only privileged mode software can access the facilities in the privileged mode environment. Problem-state software should never be allowed access to any facilities that are defined in the privileged mode environment. Hardware does not check that an access to a privileged mode facility is performed by a privileged mode process.

12.1.1 SPE Privilege 1 Facilities

Table 12-1 lists all CBEA-compliant Synergistic Processor Element (SPE) registers that are designated for privilege 1 access. For information about each of the registers, see the relevant page.

Table 12-1. SPE Privilege 1 Memory Map (Page 1 of 3)

Offset (Hexadecimal)	Register	Description	Access Type
Control and Configuration Area			
x‘0000’	MFC_SR1	MFC State Register One (see page 221)	Read/Write
x‘0008’	MFC_LPID	MFC Logical Partition ID Register (see page 223)	Read/Write
x‘0010’	SPU_ID	SPU Identification Register (see page 291)	Read/Write
x‘0018’	MFC_VR	MFC Version Register (see page 290)	Read Only
x‘0020’	SPU_VR	SPU Version Register (see page 289)	Read Only
x‘0028’:x‘00FF’	Reserved	Reserved	
Interrupt Area			
x‘0100’	INT_Mask_class0	Class 0 Interrupt Mask Register (see page 276)	Read/Write
x‘0108’	INT_Mask_class1	Class 1 Interrupt Mask Register (see page 277)	Read/Write
x‘0110’	INT_Mask_class2	Class 2 Interrupt Mask Register (see page 278)	Read/Write
x‘0118’:x‘013F’	Reserved	Reserved	Reserved
x‘0140’	INT_Stat_class0	Class 0 Interrupt Status Register (see page 280)	Read/Write
x‘0148’	INT_Stat_class1	Class 1 Interrupt Status Register (see page 281)	Read/Write
x‘0150’	INT_Stat_class2	Class 2 Interrupt Status Register (see page 282)	Read/Write
x‘0158’:x‘017F’	Reserved	Reserved	Reserved
x‘0180’	INT_Route	Interrupt Routing Register (see page 283)	Read/Write
x‘0188’:x‘01FF’	Reserved	Reserved	Reserved
Atomic Unit Control Area			
x‘0200’	MFC_Atomic_Flush	MFC Atomic Flush Register (see page 236) This is an implementation-dependent register.	Read/Write
x‘0208’:x‘03FF’	SPU_Cache_ImplRegs	Synergistic processor unit (SPU) cache hardware implementation-dependent registers. See the specific implementation documentation.	

1. An implementation should support reading of these registers for diagnostic purposes.

Table 12-1. SPE Privilege 1 Memory Map (Page 2 of 3)

Offset (Hexadecimal)	Register	Description	Access Type
Translation Lookaside Buffer (TLB) Management Registers			
x‘0400’	MFC_SDR	MFC Storage Description Register (see page 224) Also see the *PowerPC Architecture, Book III* for a description of this register.	Read/Write
x‘0408’:x‘04FF’	Reserved	Reserved	
x‘0500’	MFC_TLB_Index_Hint	TLB Index Hint Register (see page 209) Index to the best TLB entry to update.	Read Only
x‘0508’	MFC_TLB_Index	TLB Index Register (see page 210)[1] Index to the TLB entry to update with the TLB Real Page Number Register and the TLB Virtual Page Number Register.	Write Only
x‘0510’	MFC_TLB_VPN	TLB Virtual Page Number Register (see page 211) Access to the upper portion of the TLB entry.	Read/Write
x‘0518’	MFC_TLB_RPN	TLB Real Page Number Register (see page 212) Access to the lower portion of the TLB entry.	Read/Write
x‘0520’:x‘053F’	Reserved	Reserved	
x‘0540’	MFC_TLB_Invalidate_ Entry	TLB Invalidate Entry Register (see page 214)[1] Virtual page number of the TLB entry to invalidate. **Note:** Not available for the PowerPC Processor Element (PPE).	Write Only
x‘0548’	MFC_TLB_Invalidate_ All	TLB Invalidate All Register (see page 216)[1] Invalidate all TLB entries (optional). **Note:** Not available for the PowerPC Processor Element (PPE).	Write Only
x'0550':'057F’	Reserved	Reserved	
Memory Management (Implementation-dependent area: See the specific implementation documentation.)			
x‘0580’:x‘05FF’	SPE_MMU_ImplRegs	SPE Memory Management Unit (MMU) Registers See the specific implementation documentation for a description of these registers.	
Memory Flow Controller (MFC) Status and Control Area			
x‘0600’	MFC_ACCR	MFC Address Compare Control Register (see page 227)	Read/Write
x‘0610’	MFC_DSISR	MFC Data Storage Interrupt Status Register (see page 226)	Read/Write
x‘0620’	MFC_DAR	MFC Data Address Register (see page 225)	Read/Write
x‘0628’:x‘06FF’	Reserved	Reserved	
Replacement Management Table (RMT) Area (Implementation-dependent area. See the specific implementation documentation.)			
x‘0700’	MFC_TLB_RMT_Index	RMT Index Register (see page 257) Index of the replacement management tables.	Read/Write
x‘0710’	MFC_TLB_RMT_Data	RMT Data Register (see page 258) Doubleword of RMT data pointed to by the RMT Index Register. Entry contents are implementation dependent.	Read/Write
x‘0718’:x‘07FF’	SPE_RMT_ImplRegs	SPE RMT hardware implementation-dependent registers	

1. An implementation should support reading of these registers for diagnostic purposes.

Table 12-1. SPE Privilege 1 Memory Map (Page 3 of 3)

Offset (Hexadecimal)	Register	Description	Access Type
MFC Command Data Storage Interrupt Area			
x‘0800’	MFC_DSIPR	MFC Data Storage Interrupt Pointer Register (see page 232) Contains a pointer to the command in the MFC command queue that caused the error condition.	Read Only
x‘0808’	MFC_LSACR	MFC Local Storage Address Compare Register (see page 229) 64-bit MFC Local Storage Address Compare Register	Read/Write
x‘0810’	MFC_LSCRR	MFC Local Storage Compare Result Register (see page 230) 64-bit MFC Local Storage Compare Result Register	Read Only
x‘0818’:x‘08FF’	Reserved	Reserved	
Real-Mode Support Registers			
x‘0900’	MFC_RMAB	MFC Real-Mode Address Boundary Register (see page 218)	Read/Write
x‘0908’:x‘0BFF’	Reserved	Reserved	
MFC Command Error Area			
x‘0C00’	MFC_CER	MFC Command Error Register (see page 231) Contains a pointer to the command in the MFC command queue that caused the error condition.	Read Only
x‘0C08’:x‘0FFF’	Reserved	Reserved	
Implementation-Dependent Area (See the specific implementation documentation for a detailed description of these registers)			
x‘1000’:x‘1FFF’	PV1_ImplRegs	Privilege 1 implementation-dependent registers	

1. An implementation should support reading of these registers for diagnostic purposes.

12.1.2 SPE Privilege 2 Facilities

Table 12-2 lists all CBEA-compliant registers that are designated for privilege 2 access. For information about each of the registers, see the relevant page.

Table 12-2. SPE Privilege 2 Memory Map (Page 1 of 2)

Offset (Hexadecimal)	Register	Description	Access Type
MFC Registers			
x‘00000’:x‘010FF’	Reserved	Reserved	Reserved
Segment Lookaside Buffer Management Registers			
x‘01100’	Reserved	Reserved	Reserved
x‘01108’	SLB_Index	SLB Index Register (see page 201)[1] Index to the segment lookaside buffer (SLB) entry to be updated by the SLB_VSID and SLB_ESID ports.	Write Only
x‘01110’	SLB_ESID	SLB Effective Segment ID Register (see page 202) Access to the upper portion of an SLB entry.	Read/Write
x‘01118’	SLB_VSID	SLB Virtual Segment ID Register (see page 203) Access to the lower portion of an SLB entry.	Read/Write
x‘01120’	SLB_Invalidate_Entry	SLB Invalidate Entry Register (see page 205)[1] Effective segment ID (ESID) of the SLB entry to invalidate.	Write Only
x‘01128’	SLB_Invalidate_All	SLB Invalidate All Register (see page 206)[1] Invalidate all SLB entries.	Write Only
x‘01130’:x‘01FFF’	Reserved	Reserved	Reserved
Context Save and Restore Area. (Implementation-dependent area: See the specific implementation documentation.)			
x‘02000’:x‘02FFF’	MFC_CSR_ImplRegs	MFC Context Save and Restore registers	
MFC Control			
x‘03000’	MFC_CNTL	MFC Control Register (see page 233)	Read/Write
x‘03008’:x‘03FFFF’	MFC_Cntl1_ImplRegs	Implementation-dependent control registers. See the specific implementation documentation.	
Interrupt Mailbox			
x‘04000’	SPU_Out_Intr_Mbox	SPU Outbound Interrupt Mailbox Register (see page 237) SPU writes; PPE reads.	Read Only
SPU Control			
x‘04040’	SPU_PrivCntl	SPU Privileged Control Register (see page 239)	Read/Write
x‘04058’	SPU_LSLR	SPU Local Storage Limit Register (see page 241)	Read/Write
x‘04060’	SPU_ChnlIndex	SPU Channel Index Register (see page 242) This register selects which SPU channel in the specified SPU(n) is accessed using the SPU Channel Count Register or the SPU Channel Data Register.	Read/Write
x‘04068’	SPU_ChnlCnt	SPU Channel Count Register (see page 244) This register reads or initializes the SPU Channel Count Register selected by the SPU Channel Index Register.	Read/Write

1. An implementation should support reading of these registers for diagnostic purposes.

Table 12-2. SPE Privilege 2 Memory Map (Page 2 of 2)

Offset (Hexadecimal)	Register	Description	Access Type
x‘04070’	SPU_ChnlData	SPU Channel Data Register (see page 243) This register reads or initializes the SPU channel data selected by the SPU Channel Index Register.	Read/Write
x‘04078’	SPU_Cfg	SPU Configuration Register (see page 245) This register is used to read or set the configuration of the SPU Signal-Notification Registers in the specified SPU(n).	Read/Write
x‘04080’:x‘04FFF’	Reserved	Reserved	
Implementation-Dependent Area. (See the specific implementation documentation for a detailed description of these registers).			
x‘05000’:x‘0FFFF’	PV2_ImplRegs	Privilege 2 implementation-dependent registers	
Reserved Area			
x‘10000’:x‘1FFFF’	Reserved	Reserved	

1. An implementation should support reading of these registers for diagnostic purposes.

12.1.3 PPE Privilege 1 Facilities

Table 12-3 lists all CBEA-compliant PPE registers that are designated for privilege 1 access. For information about each of the registers, see the relevant page.

Table 12-3. PPE Privilege 1 Memory Map

Offset (Hexadecimal)	Register	Description	Access Type
Replacement-Management Table (RMT) Area (Implementation-dependent area: See the specific implementation documentation.)			
x‘200’:x‘2FF’	Reserved	Reserved	
x‘300’	L2_RMT_Index	RMT Index Register (see page 257) Index of the replacement-management tables.	Read/Write
x‘310’	L2_RMT_Data	RMT Data Register (see page 258) Doubleword of RMT data pointed to by the RMT Index Register. Entry contents are implementation dependent.	Read/Write
x‘318’:x‘7FF’	Reserved	Reserved	
Implementation-Dependent Area (See the specific implementation documentation for a detailed description of these registers.)			
x‘800’:x‘FFF’	PPEPV_ImplRegs	PPE privilege 1 implementation-dependent registers.	

13. PowerPC Architecture, Book III Compatibility

This section covers the compatibility of the PowerPC Processor Element (PPE) facilities used in a CBEA-compliant system with the privileged facilities defined in *PowerPC Architecture, Book III*.

13.1 Optional Features in PowerPC Architecture, Book III (Required for CBEA)

The following facility is considered optional in the *PowerPC Architecture, Book III* but are required for compliance with the Cell Broadband Engine Architecture (CBEA).

- Real-mode storage control (see *Section 14.4 Real-Mode Storage Control Facilities* on page 217 for more information).

13.2 Incompatibilities with PowerPC Architecture, Book III

There are no incompatibilities with *PowerPC Architecture, Book III*.

13.3 Extensions to the PowerPC Architecture

See *Appendix E Extensions to the PowerPC Architecture* on page 315 for a description of the following instructions and facilities that are not described in the *PowerPC Architecture* documents.

- Software Management of TLBs (optional) (see page 315)
- Mediated External Exception Extension (optional) (see page 316)
- Multiple Concurrent Large Pages (optional) (see page 318)
- Defined Behavior for Inaccessible SPRs (see page 319)

14. Storage Addressing

The storage addressing of the Cell Broadband Engine Architecture (CBEA) is compatible with the PowerPC Architecture. An address translation mechanism processes an effective address provided by an instruction or a memory flow controller (MFC) command. The mechanism for a PowerPC Processor Element (PPE) is described in *PowerPC Architecture, Book III.* The memory management unit (MMU) of the Synergistic Processor Element (SPE) employs the same basic mechanism to process an effective address provided by an MFC command. The PowerPC address translation mechanism has been extended to support multiple concurrent large pages and the software management of translation lookaside buffers (TLBs) (see *Section E.3 Multiple Concurrent Large Pages (optional)* on page 318).

If the Instruction Relocate (IR) and the Data Relocate (DR) bits in the PowerPC Machine State Register are set to '1', address translations are enabled for instructions and data fetches by the PPEs. To enable translations of the effective addresses used in MFC commands, set the Relocate (R) bit of the MFC State Register One (see page 221) to '1'.

Two steps are required to convert an effective address to a real address:

1. **Convert the effective address to a virtual address.** The conversion to a virtual address uses a segment lookaside buffer (SLB). The MMU for an SPE differs from the PowerPC Architecture only in the minimum number of SLB entries that must be provided by an implementation. The PowerPC Architecture requires a minimum of 64 entries; an SPE MMU requires a minimum of eight SLB entries. The maximum number of SLB entries is 4 K. All implementations must have at least a minimum number of SLB entries and the associated management instructions, special purpose registers (SPRs), and memory-mapped I/O (MMIO) registers.
2. **Convert the virtual address to a real address.** The conversion of a virtual address to a real address uses a page table in main storage. The page table format and the conversion process are described in *PowerPC Architecture, Book III*. In the software-managed mode, the TLB management instructions and registers must provide the capability to directly specify a virtual-address-to-real-address translation without using a hardware-accessed page table in main storage.

To enhance performance of these conversions, most implementations provide translation lookaside buffer management (see *Section 14.3* on page 207). The TLB is, basically, a special cache for keeping the recently used page table entries (PTEs). When operating in hardware-accessed page table mode, the TLB need not be kept consistent with the hardware-accessed page table in main storage. All implementations must support both a virtual-to-real-address translation mechanism using a hardware-accessed page table in main storage and a software-managed translation mechanism. The software-managed translation mechanism uses the translation lookaside buffer management facilities to directly supply the translations without the need for a hardware-accessed page table in main storage. (For more information, see *Section 14.3 Translation Lookaside Buffer Management* beginning on page 207.)

Privileged software must manage the SLB in the CBEA for all PPE and SPE units that use address translation. The TLB is managed by hardware accesses to a page table in main storage in the PowerPC Architecture. The CBEA provides both a hardware-managed TLB mode and a privileged software-managed TLB mode. For both a PPE and an SPE, software management of the TLB allows the system software designer to forgo the requirement of a hardware-accessible page table and use any format for the system page table. The format and size of the page table are not restricted by hardware, as required by the PowerPC Architecture. For more information about the hardware TLB management method, see *PowerPC Architecture, Book III*.

In addition, the software management facility can be used in combination with the hardware TLB management facility to preload translations into the TLB. For information about the TLB, see *Section 14.3 Translation Lookaside Buffer Management* beginning on page 207.

14.1 PPE Segment Lookaside Buffer Management

PPE SLBs are software managed. To manage the PPE SLBs, the PowerPC Architecture supports five instructions (**slbie**, **slbia**, **slbmte**, **slbmfev**, and **slbmfee**). See *PowerPC Architecture, Book III* for a description of these instructions.

14.2 SPE Segment Lookaside Buffer Management

The CBEA provides a set of MMIO registers to manage the SLBs in an SPE. These MMIO registers provide the same functions for an SPE MMU as the PowerPC instructions provide for a PPE MMU. That is, the following registers mimic the source operands of the PowerPC SLB management instructions (**slbie**, **slbia**, **slbmte**, **slbmfev**, and **slbmfee**):

- SLB Invalidate Entry Register
- SLB Invalidate All Register
- SLB Index Register
- SLB Virtual Segment ID Register
- SLB Effective Segment ID Register

14.2.1 SLB Mapping

The SLB management registers for each SPE are mapped into the MMIO space of the main storage domain. The SLB management area must be accessed with storage attributes of caching inhibited and guarded. It should be restricted to privileged code. Software must perform the following sequence in order to replace an SLB entry:

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 (see page 205) to invalidate the SLB entry.
2. Set the new contents for the virtual segment ID (VSID) portion of the SLB entry.
3. Set the new contents for the effective segment ID (ESID) portion of the SLB entry along with the Valid bit.

The contents of an SLB entry in an SPE MMU are accessed by using the SLB Effective Segment ID Register (see page 202) and the SLB Virtual Segment ID Register (see page 203). The SLB Index Register (see page 201) points to the SLB entry to be accessed by the SLB_ESID and SLB_VSID registers. The size and format of the SLB are implementation dependent.

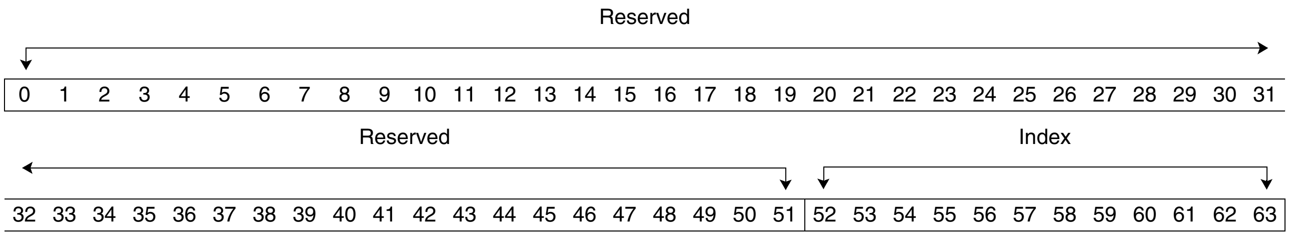
14.2.2 SLB Index Register (SLB_Index)

The SLB Index Register is used to select which SLB entry to access using the SLB_ESID and SLB_VSID registers. The SLB index is a 12-bit value. The number of index bits used and the organization of the SLB are implementation dependent. Some implementations can require software to use a specific set of indexes for a given SLB value. See the specific implementation documentation for more information.

This register can be written using a single 64-bit store operation or a single 32-bit store operation to the lower 32-bits of this register (that is, offset x‘0110C’).

Access Type Write

Base Address Offset (BP_Base | P2(n)) + x‘01108’, where n is an SPE number.



Bits	Field Name	Description
0:51	Reserved	Set to zeros.
52:63	Index	Index. The number of bits in this field is implementation dependent.

14.2.3 SLB Effective Segment ID Register (SLB_ESID)

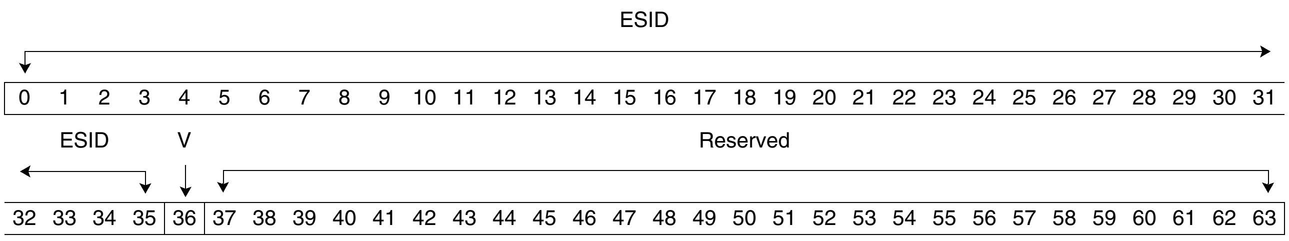
The SLB Effective Segment ID Register is used to access the upper portion of an SLB entry. The SLB_ESID contains the effective segment ID and a bit that indicates if the SLB entry selected by the SLB Index Register is valid.

This register can be written using a single 64-bit store operation or two 32-bit store operations.[1] When using 32-bit operations, the first store must be to the most-significant word. The second store must be to the least-significant word.

Note: Some implementations can support a cache of effective-to-real-address translations (ERATs) to improve performance. Setting the valid bit to ‘0’ does not invalidate any cached translations of the SLB entry. The SLB Invalidate Entry Register must be used for this purpose.

Address Read/Write

Base Address Offset (BP_Base | P2(n)) + x‘01110’, where n is an SPE number.



Bits	Field Name	Description
0:35	ESID	Effective segment ID.
36	V	When set, indicates that the SLB entry is valid.
37:63	Reserved	Set to zeros.

1. In general, 64-bit access to an address range that includes a 32-bit MMIO register is not allowed except when specified explicitly as in this instance.

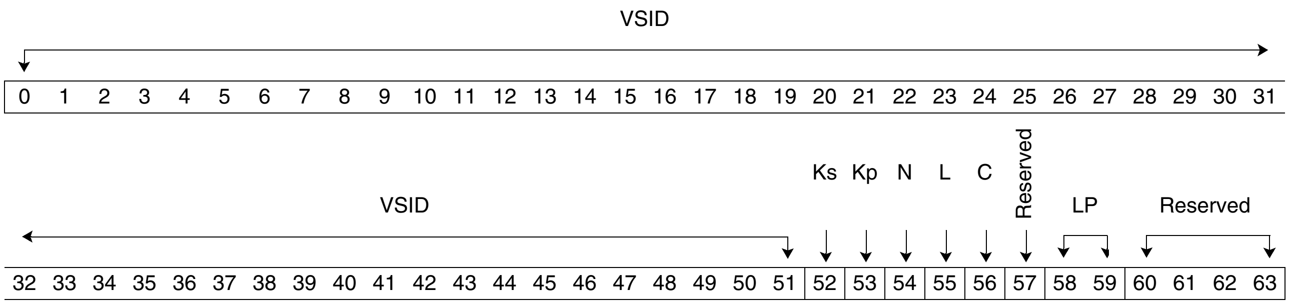
14.2.4 SLB Virtual Segment ID Register (SLB_VSID)

The SLB Virtual Segment ID Register is used to access the lower portion of an SLB entry. The SLB_VSID contains the virtual segment ID and other miscellaneous characteristics of the memory segment. The SLB entry is selected by the SLB Index Register (see page 201).

This register can be written using a single 64-bit store operation or two 32-bit store operations.[1] When using 32-bit operations, the first store must be to the most-significant word. The second store must be to the least-significant word.

Access Type Read/Write

Base Address Offset (BP_Base | P2(n)) + x‘01118’, where n is an SPE number.



Bits	Field Name	Description
0:51	VSID	Virtual segment ID.
52	Ks	Supervisor (privileged) state storage key This value, along with the Kp bit, is used to compute a key that is used with the page protection (PP) bits in the PTE for storage protection.
53	Kp	Problem-state storage key This value, along with the Ks bit, is used to compute a key that is used with the PP bits in the PTE for storage protection.
54	N	No execute segment 0 Instruction fetches are allowed. 1 Instruction fetches are not allowed. This is bit is ignored by the MFC.
55	L	Virtual page-size selector The size of the large page selected by the SLB is the L bit concatenated with the LP field (L || LP). The page sizes supported are implementation dependent. The number of concurrent page sizes supported is also implementation dependent. If an implementation supports more page sizes than can be concurrently supported by L || LP, a configuration mechanism should be provided to assign a page size to an L || LP value. This configuration should not change frequently. Therefore, the configuration mechanism does not need to be directly accessible by privileged software. The page size selected by L || LP must match the page size in the PTE when searching the page table. (See *Appendix E* on page 315 for more details.)
56	C	Class The class field is used in conjunction with the SLB Invalidate Entry Register (see page 205). It is used as an additional qualifier for the ESID when multiple virtual address spaces exist.
57	Reserved	Set to zero.

1. In general, 64-bit access to an address range that includes a 32-bit MMIO register is not allowed except when specified explicitly as in this instance.

Bits	Field Name	Description
58:59	LP	Virtual page-size selector The size of the large page selected by the SLB is the L bit concatenated with the LP field (L || LP). The page sizes supported are implementation dependent. The number of concurrent page sizes supported is also implementation dependent. If an implementation supports more page sizes than can be concurrently supported by L || LP, a configuration mechanism should be provided to assign a page size to an L || LP value. This configuration should not change frequently. Therefore, the configuration mechanism does not need to be directly accessible by privileged software. The page size selected by L || LP must match the page size in the PTE when searching the page table. (See *Appendix E* on page 315 for more details.)
60:63	Reserved	Set to zeros.

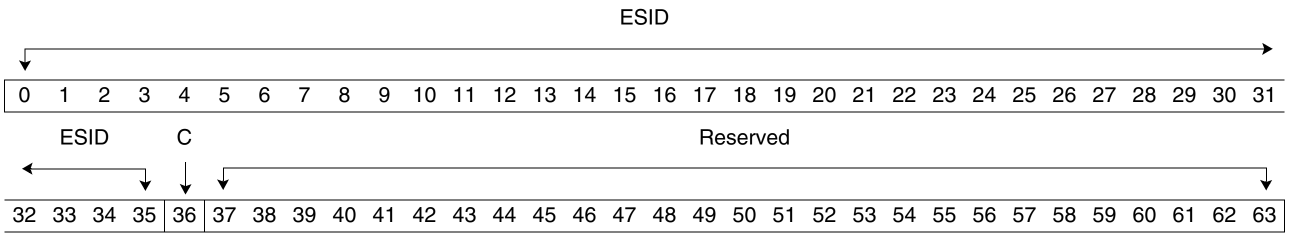
14.2.5 SLB Invalidate Entry Register (SLB_Invalidate_Entry)

This register can be written using a single 64-bit store operation or two 32-bit store operations.[1] When using 32-bit operations, the first store must be to the most-significant word. The second store must be to the least-significant word.

Any write to the least-significant word causes an entry in the SLB to be invalidated. Writing only the least-significant word will invalidate an entry that matches the upper bits of the SLB_ESID value (contained in the most-significant word) concatenated with the lower bits of the SLB_ESID (provided by data written to the least-significant word).

Address Write

Base Address Offset (BP_Base | P2(n)) + x'01120', where n is an SPE number.



Bits	Field Name	Description
0:35	ESID	Effective segment ID.
36	C	Class. The class field is used in conjunction with the SLB Invalidate Entry Register. It is used as an additional qualifier for the ESID when multiple virtual address spaces exist.
37:63	Reserved	Set to zeros.

1. In general, 64-bit access to an address range that includes a 32-bit MMIO register is not allowed except when specified explicitly as in this instance.

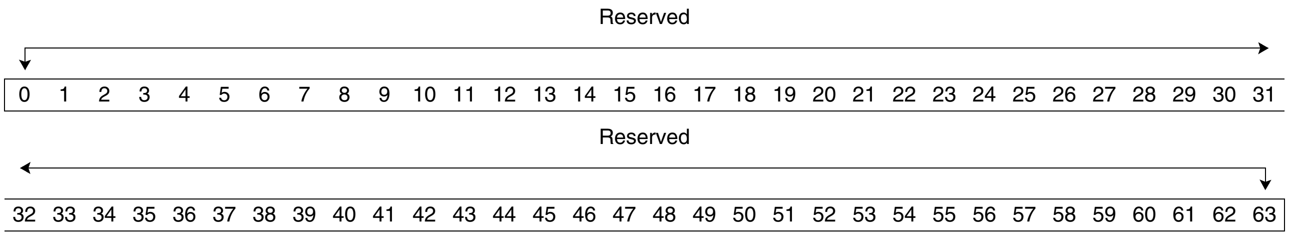
14.2.6 SLB Invalidate All Register (SLB_Invalidate_All)

A write to the SLB Invalidate All Register causes the Valid (V) bit in all entries of the SLB to be set to ‘0’, making the entries invalid. The remaining fields of each entry are undefined.

This register can be written using a single 64-bit store operation or two 32-bit store operations.[1] Any write to the least-significant word causes all entries in the SLB to be invalidated.

Access Type Write

Base Address Offset (BP_Base | P2(n)) + x‘01128’, where n is an SPE number.



Bits	Field Name	Description
0:63	Reserved	Set to zeros. The data for this register is reserved for future use. Writing to this register causes the contents of the segment lookaside buffer to be voided.

Implementation Note:

The SLB Invalidate All Register for an SPE MMU clears the V bit of SLB entry 0. This differs from the PowerPC **slbia** instruction, which does not clear the V bit of SLB entry 0.

1. In general, 64-bit access to an address range that includes a 32-bit MMIO register is not allowed except when specified explicitly as in this instance.

14.3 Translation Lookaside Buffer Management

The translation lookaside buffer is an on-chip cache that stores page-table entries for the most recently accessed pages. It enhances performance by eliminating the need to access the page table from memory during load-store operations.

The CBEA permits both a hardware TLB reload and a software TLB reload when a translation is not found in the TLB table. For an SPE, the TLB reload mode is selected by the Load Control (TL) bit of the MFC State Register One (see page 221). For a PPE, the TLB reload mode is selected by the TL bit of the Logical Partitioning Control Register (LPCR[53]). For more information, see the specific implementation documentation.

Note: Software management of the TLB is a CBEA extension to the PowerPC Architecture (see *Section E Extensions to the PowerPC Architecture* on page 315).

One difference between a hardware TLB reload and a software TLB reload is the point at which the data storage interrupt (DSI) or SPE interrupt is presented to a PPE. For a software TLB reload, the interrupt is generated when a translation is not found in the TLB. For a hardware TLB reload, the interrupt is generated only after the page table is searched by hardware and a translation is not found. The TLB management facilities function identically for hardware-managed and software-managed TLB reloads.

Programming Note:

When updating a TLB entry in a PPE or SPE that is set to operate in software TLB reload mode, software should set both the Reference (R) bit and the Change (C) bit in the TLB Real Page Number Register (see page 212). Software cannot rely on an interrupt when an access is performed to a page whose corresponding TLB entry has the R bit set to zero, or when a write (store or direct memory access [DMA] put) is performed to a page whose corresponding TLB entry has the C bit set to zero. Software should use the page protection bits in the TLB to manage the R and C bits in the software page table.

Implementation Note:

An implementation can choose how to handle the following cases when operating in software TLB reload mode:

- The R bit is '0', and an access is performed to the page.
- The C bit is '0', and a write operation is performed to the page.

The implementation can choose one of the following actions:

- Ignore the R and C bits.
- Generate a data storage interrupt.
- Update the R and C bits in the TLB.

The hardware-accessed page table is a variably sized data structure that specifies the mapping between virtual page numbers and real page numbers. Each PTE maps one virtual page number to one real page number. The hardware-accessed page table search is defined in *PowerPC Architecture, Book III*. If the translation is not found in the hardware-accessed page table because of a page or mapping fault, either a DSI (for a PPE) or an interrupt (for an SPE) is posted to the PPE.

In software TLB reload mode, the format of the page table is software dependent. The table is not accessed by hardware. When a DSI or an SPE interrupt is posted indicating a translation is not available in the TLB, software should perform a page table search to resolve the TLB miss. In this mode, the MFC Storage Description Register is not used.

For an SPE TLB, a switch between software and hardware management is made by setting the TL bit in MFC State Register One (see page 221). To switch SPE TLB management modes, software should suspend the MFC command queue operation and issue a **sync** instruction before the switch. Switching modes in a PPE is implementation dependent. The registers defined in this section provide the same function for both a PPE and an SPE. The PPE registers are located in the SPR space (see *Table C-1* on page 309), and the SPE registers are located in the CBEA memory map (see *Table A-2* on page 296). The effective address of the operation that causes a translation fault in a PPE is placed in the PPE Data Address Register, defined in the PowerPC Architecture. The effective address of the operation that causes a translation fault in the MFC is placed in the MFC Data Address Register (see page 225). The MFC Data Address Register is independent for each MFC, and the PPE Data Address Register is independent for each PPE.

14.3.1 TLB Mapping

The MFC TLB area must be marked caching inhibited and guarded. It should be restricted to privileged mode code.

In software TLB management mode, hardware must set the TLB Index Hint Register (see page 209) to the TLB entry selected for replacement. Software can change the selected entry as long as the new entry is within the same congruency class. Software must perform the following sequence of operations to replace a TLB entry:

1. Set the index of the TLB entry to be replaced.
2. Set the new contents for the real page number (RPN) portion of the TLB entry.

   Note: A write to the RPN portion of the TLB does not cause the TLB entry to be updated.
3. Set the new contents for the virtual page number (VPN) portion of the TLB entry along with the Valid bit.

   Note: A write to the VPN portion of the TLB causes the TLB entry to be updated.

Programming Note:

Depending upon the implementation, invalidating the TLB entry using a **tlbie** (local) instruction or the TLB Invalidate Entry Register (see page 214) can cause more than one TLB entry to be invalidated (a congruency class, for example). When the TLB is software managed, privileged software must keep a shadow copy of the state of the TLB table. Software must reenable the entries that were not meant to be invalidated.

The size and format of the TLB are implementation dependent. The TLB contents are described in the specific implementation documentation.

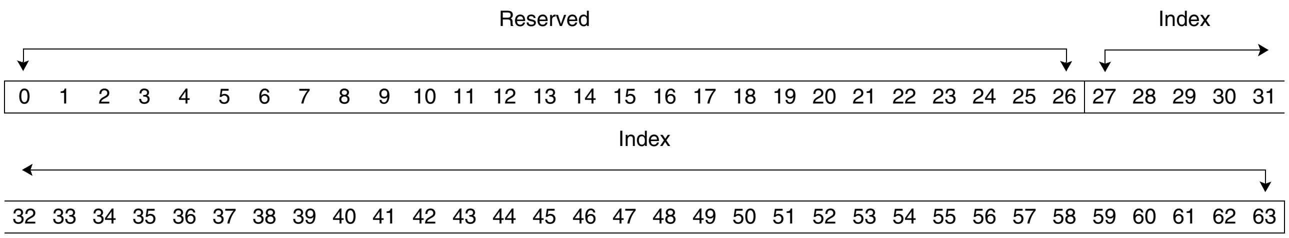
14.3.2 TLB Index Hint Register (TLB_Index_Hint)

This register is only used in software management mode. This register is an SPR in a PPE and is accessible using an **mfspr** instruction. This register is mapped into MMIO space for an SPE.

Hardware must set the TLB Index Hint Register to the TLB entry selected for replacement when a translation fault occurs. When software manages the TLB, the TLB Index Hint Register is used to determine which TLB entry the hardware algorithm selected for replacement. Software can read the TLB Index Hint Register and use this value as the TLB Index Register, or it can select a new index within the same congruency class. The TLB Index Hint Register is useful because software cannot always determine the best entry for replacement (that is, the least recently used). See the specific implementation documentation for more information about the TLB Index Hint Register.

The TLB Index Hint Register is a separate register from the TLB Index Register (see page 210) to prevent hardware from changing the index if a fault occurs while software is updating a TLB entry.

Access Type	Read Only
Base Address Offset	(BP_Base | P1(n)) + x‘0500’, where n is an SPE number.
PPE SPR Number	x‘3B2’



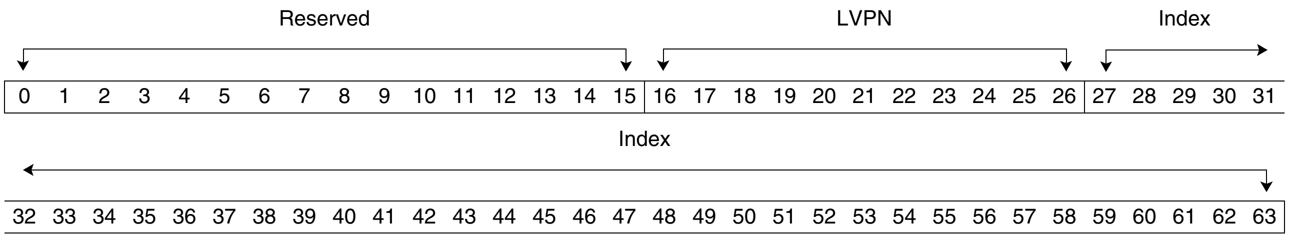
Bits	Field Name	Description
0:26	Reserved	Set to zeros by hardware.
27:63	Index	Index. This field contains information that is used by an implementation to update a TLB entry. The TLB index in the Index field is a function of the virtual address. The function is implementation-dependent. The number of bits in this field is also implementation dependent.

14.3.3 TLB Index Register (TLB_Index)

This register is only used in software management mode. This register is an SPR in a PPE and is accessible using either an **mfspr** or an **mtspr** instruction. This register is mapped into MMIO space for an SPE.

The TLB Index Register is used to select the entry in the TLB that will be modified by the TLB Virtual Page Number Register (see page 211) and the TLB Real Page Number Register (see page 212).

Access Type	Write
Base Address Offset	(BP_Base I P1(n)) + x‘0508’, where n is an SPE number.
PPE SPR Number	x‘3B3’



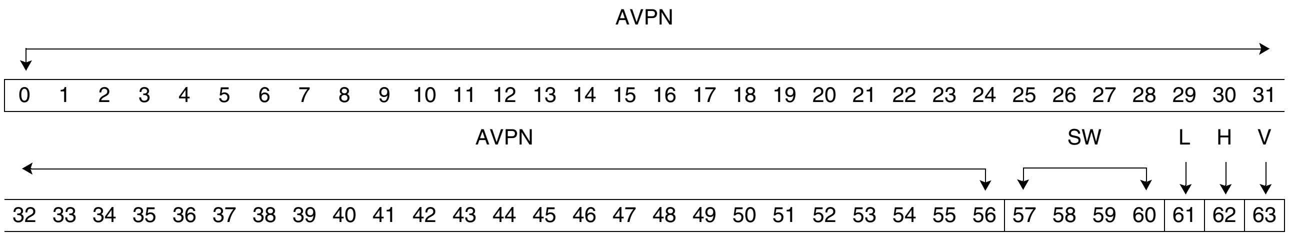
Bits	Field Name	Description
0:15	Reserved	Set to zeros.
16:26	LVPN	Lower virtual page number. These are the lower 11 bits of the virtual page number (the abbreviated VPN from the TLB Virtual Page Number Register (see page 211) concatenated with the LVPN yields the VPN).
27:63	Index	This field contains information used by an implementation to update a TLB entry. The TLB index in the Index field is a function of the virtual address. The function is implementation-dependent. The number of bits in this field is also implementation dependent.

14.3.4 TLB Virtual Page Number Register (TLB_VPN)

This register is only used in software management mode. This register is an SPR in a PPE and is accessible using either an **mfspr** or an **mtspr** instruction. This register is mapped into MMIO space for an SPE.

The TLB Virtual Page Number Register provides access to the upper 64 bits (or the virtual page number portion) of the TLB entry. For more information, see the description of the page table entry in *PowerPC Architecture, Book III.*

Access Type	Read/Write
Base Address Offset	(BP_Base l P1(n)) + x‘0510’, where n is an SPE number.
PPE SPR Number	x‘3B4’



Bits	Field Name	Description
0:56	AVPN	Abbreviated virtual page number.
57:60	SW	An optional field available for software use.
61	L	Virtual page-size indicator. 0 4 KB 1 Virtual pages are large. The size of the page is controlled by the LP field in the TLB Real Page Number Register (see page 212).
62	H	Hash function identifier. 0 Primary hash 1 Secondary hash
63	V	0 Entry invalid 1 Entry valid

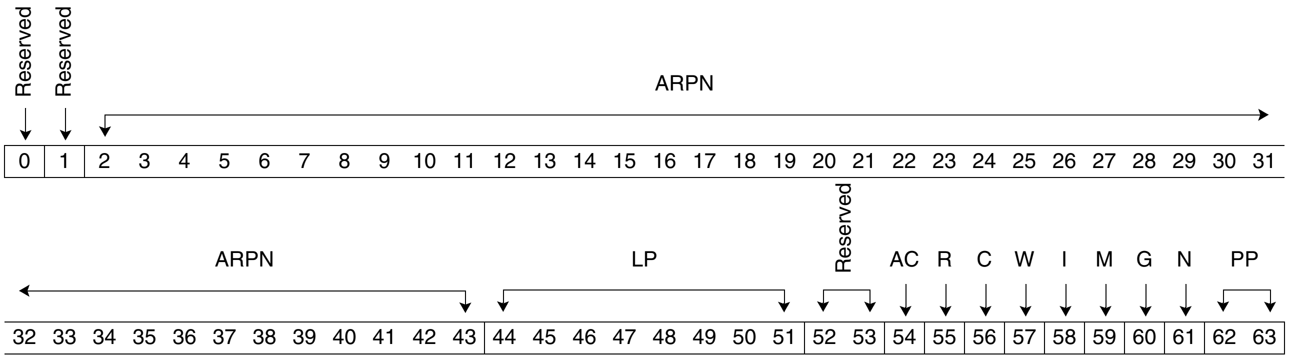
Note: If the VPN is invalidated to change the protection attributes of a page, or to “steal” a page, a TLB invalidate entry command must be issued to invalidate any cache of the effective-to-real-address translation that can be associated with the TLB entry being invalidated.

14.3.5 TLB Real Page Number Register (TLB_RPN)

This register is only used in software management mode. This register is an SPR in a PPE and is accessible using either an **mfspr** or an **mtspr** instruction. This register is mapped into MMIO space for an SPE.

The TLB Real Page Number Register provides access to the lower 64-bits (or the real page number portion) of the TLB entry. For more information, see the description of the page table entry in *PowerPC Architecture, Book III*.

Access Type	Read/Write
Base Address Offset	(BP_Base I P1(n)) + x‘0518’, where n is an SPE number.
PPE SPR Number	x‘3B5’



Bits	Field Name	Description
0	Reserved	Set to zeros.
1	Reserved	Set to zeros.
2:43	ARPN	Abbreviated real page number
44:51	LP	Size selector for a large virtual page. This field is used to select the size of a large page from the hardware-defined list of page sizes. This field supports up to eight concurrent large page sizes. The size of a page selected by the LP field is implementation dependent. The number of concurrent page sizes supported is also implementation dependent. Depending on the page size, the bits in this field represented by “a” (those to the left of the first zero) might be concatenated with the ARPN field to form the RPN for a successful lookup. **aaaaaaaa** MFC_TLB_VPN[L] = ‘0’. **aaaaaaa0** If MFC_TLB_VPN[L] = '1', large page size one is selected. **aaaaaa01** If MFC_TLB_VPN[L] = '1', large page size two is selected. ... **01111111** If MFC_TLB_VPN[L] = '1', large page size eight is selected.
52:53	Reserved	Set to zeros.
54	AC	Address compare bit. 0 Data address compare disabled for the corresponding virtual page. 1 Data address compare enabled for the corresponding virtual page.
55	R	Reference bit. The effects of this bit in software management mode are implementation dependent.
56	C	Change bit. The effects of this bit in software management mode are implementation dependent.
57	W	Write-through storage control bit.
58	I	Caching-inhibit storage control bit.

Bits	Field Name	Description
59	M	Memory-coherency storage control bit.
60	G	Guarded storage control bit.
61	N	No execute page if N = ‘1’.
62:63	PP	Page protection bits.

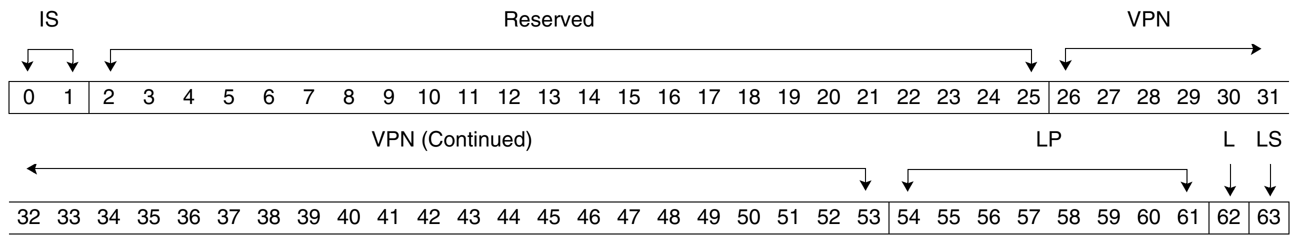
14.3.6 TLB Invalidate Entry Register (TLB_Invalidate_Entry)

This register is only used in software management mode or when privileged software preloads a TLB. This register is not available for a PPE. A PPE uses a **tlbie** or **tlbiel** instruction to invalidate TLBs. This register is mapped into MMIO space for an SPE.

The TLB Invalidate Entry Register is used to invalidate TLB entries in the MFC. The function of this register is similar to the PowerPC **tlbie** instruction. Access to this register should be privileged.

The TLB Invalidate Entry Register contains a VPN field and an IS field. The VPN is used to identify the particular entry to invalidate. The IS field is used to control how selective the invalidate should be.

Access Type	Write
Base Address Offset	(BP_Base I P1(n)) + x‘0540’, where n is an SPE number.
PPE SPR Number	No corresponding SPR number exists. A PPE uses the local form of the **tlbie** instruction.



Bits	Field Name	Description
0:1	IS	Invalidation selector. 00 The TLB is as selective as possible in invalidating the TLB entry. The implementation should use as many VPN bits as possible to eliminate invalidating unnecessary entries. 01 The TLB entry is not invalidated. Any lower-level caches of the translation are invalidated. 10 The TLB does a congruency-class invalidate if the logical-partition ID (LPID) matches the current value in the MFC Logical Partition ID Register (see page 223). 11 The TLB does a congruency-class invalidate regardless of LPID match. **Implementation Note**: An implementation can choose to implement a subset of these options. It is always acceptable for an implementation to invalidate more TLB entries than specified by this field. The IS bits are only a useful hint for a performance benefit.
2:25	Reserved	Set to zeros.
26:53	VPN	Bits 32:59 of the virtual address.
54:61	LP	Size selector for a large virtual page. This field is used to select the size of the large page from the hardware-defined list of page sizes. The size of a page selected by the LP field is implementation dependent. The number of concurrent page sizes supported is also implementation dependent. Depending on the page size selected, the bits in this field represented by “a” (those to the left of the first zero) might be concatenated with the VPN field to determine which entries are invalidated. **aaaaaaaa** TLB_Invalidate_Entry[L] = ‘0’. **aaaaaaa0** If TLB_Invalidate_Entry[L] = '1', large page size one is selected. **aaaaaa01** If TLB_Invalidate_Entry[L] = '1', large page size two is selected. ... **01111111** If TLB_Invalidate_Entry[L] = '1', large page size eight is selected. Software should set the least-significant bit of the LP field to the same value as the LS bit for compatibility with implementations that only support two large page sizes.

Bits	Field Name	Description
62	L	Large Page indicator 0 Page is small (4 KB). 1 Page is large. See LP field.
63	LS	Large Page Selection 0 First large page (The size is implementation dependent.) 1 Second large page (The size is implementation dependent.) **Note:** Software should set this bit to the same value as the least-significant bit of the LP field for compatibility with implementations that only support two large page sizes.

Programming Note:

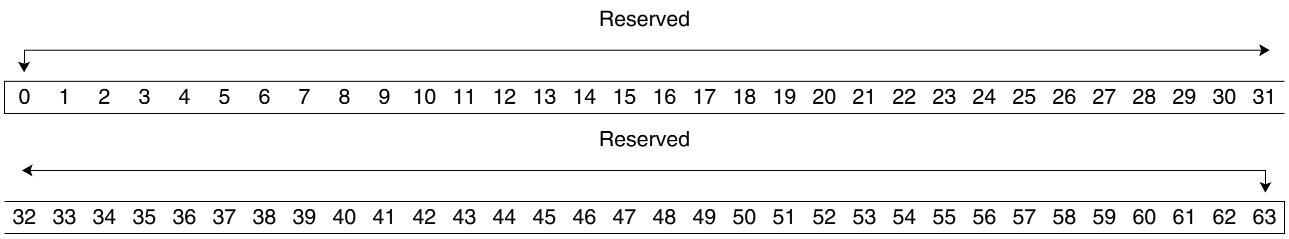
1. If the VPN is being invalidated to change the protection attributes of a page, or to “steal” a page, a TLB Invalidate Entry command must be issued to invalidate any cache of the effective-to-real-address translation that can be associated with the TLB entry being invalidated. The IS field in the TLB Invalidate Entry command is used to invalidate the cache without affecting TLB entries.
2. Care must be taken in using this function in real-time or TLB-managed environments, because hardware can invalidate all TLB entries in the associated congruency class. This could adversely affect TLB set management and real-time deterministic response. To avoid this side effect for real-time environments, privileged software can use the TLB index and TLB direct modification functions to locate the specific entry to be invalidated in the congruency class and only invalidate the entry that matches.

14.3.7 TLB Invalidate All Register (TLB_Invalidate_All)

This register is only used in software management mode or when privileged software preloads a TLB. This register is not available for a PPE. A PPE uses a **tlbia** instruction to invalidate TLBs. This register is mapped into MMIO space for an SPE.

The TLB Invalidate All Register is used to invalidate all TLB entries in the MFC command queues. The function of this register is similar to the PowerPC **tlbia** instruction. Access to this register should be privileged.

Access Type	Write
Base Address Offset	(BP_Base | P1(n)) + x‘0548’, where n is an SPE number.
PPE SPR Number	No corresponding SPR number exists. A PPE uses the local form of the **tlbia** instruction.



Bits	Field Name	Description
0:63	Reserved	Set to zeros. The data for this register is reserved for future use. The writing to this register causes the contents of the TLB to be voided.

14.4 Real-Mode Storage Control Facilities

14.4.1 PPE Real-Mode Storage Control Facility

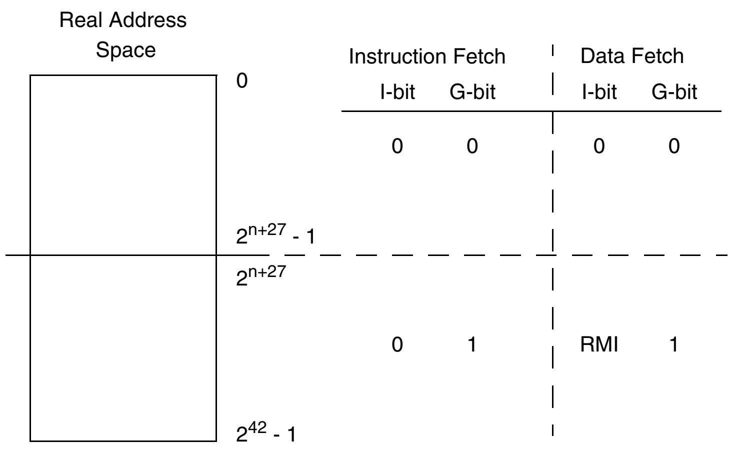
The PPE real-mode storage control facility in the PowerPC processor is an optional facility in the PowerPC Architecture. A PPE supports a 4-bit real-mode storage control (RMSC) field in an implementation-dependent register.

The RMSC field is used to control the guarded storage control attribute when a PPE is running with translation off (that is, PowerPC MSR[IR] = '0' and MSR[DR] = '0'). Access to this facility should be privileged. The RMSC field provides a mechanism for setting the boundary between storage that is considered well behaved and storage that is not. The boundary, as illustrated in *Figure 14-1*, is a power of 2 from 256 MB up to 4 TB for $1 \le n \le 15$. The RMSC field has no effect when a PPE is running with translation on (that is, PowerPC MSR[IR] = '1' and MSR[DR] = '1').

If the RMSC field is set to a value of 'n,' all accesses within the first $2^{(n+27)}$ bytes, for $1 \le n \le 15$ of the real address space are considered well behaved and cacheable. Memory that is well behaved typically has the guarded (G) attribute set to zero. The caching inhibited (I) attribute can be either '0' or '1' but in real mode is typically set to '0'. Data accesses outside the first $2^{(n+27)}$ bytes of the real address space can be neither well behaved nor cacheable. In this case, the guarded (G) attribute is set to '1'. Caching is controlled by the Real Mode Caching Inhibited (RMI) bit in an implementation-dependent PowerPC register in real addressing mode.

When the RMSC field is set to a value of '0', no data accesses are considered well behaved and caching is controlled by the RMI bit (that is, I = RMI, G = '1') in real addressing mode. All instruction fetches are not well behaved but are cacheable (that is, the caching inhibited attribute, I, equals zero, and the guarded attribute, G, equals '1') in real addressing mode.

Figure 14-1. Real-Mode Storage Boundary (showing instruction fetches and data fetches)



14.4.2 MFC Real-Mode Address Boundary Register (MFC_RMAB)

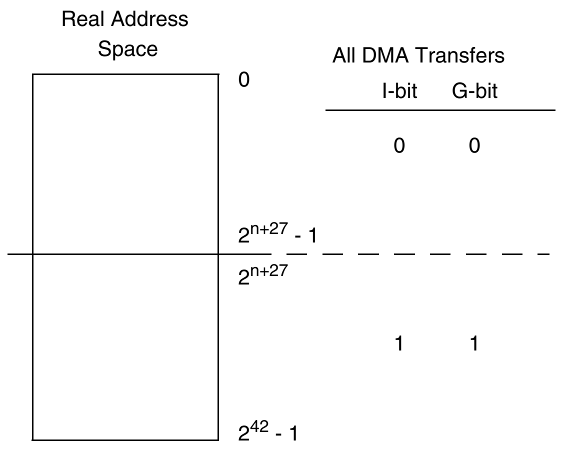
The MFC real-mode address boundary facility in the MFC is an optional facility in the CBEA. The MFC supports a 4-bit real-mode boundary (RMB) field in this implementation-dependent register.

The RMB field is used to control the caching inhibited (I) and guarded (G) storage control attributes when the MFC is running with translation off, which means that real-mode addressing is on (that is, MFC_SR1[R] = '0'). See *MFC State Register One* on page 221 for more information. The 4 lower bits of this register provide a mechanism for setting the boundary between storage that is considered well behaved and cacheable and storage that is not. The boundary, as illustrated in *Figure 14-2*, is a power of 2 from 256 MB up to 4 TB for $1 \leq n \leq 15$. The RMB field has no effect when the MFC is running with translation on (that is, when the Relocate (R) bit is set to '1' in the MFC State Register One).

For this operation to work properly, privileged software must suspend all MFC operations before modifying the contents of this register.

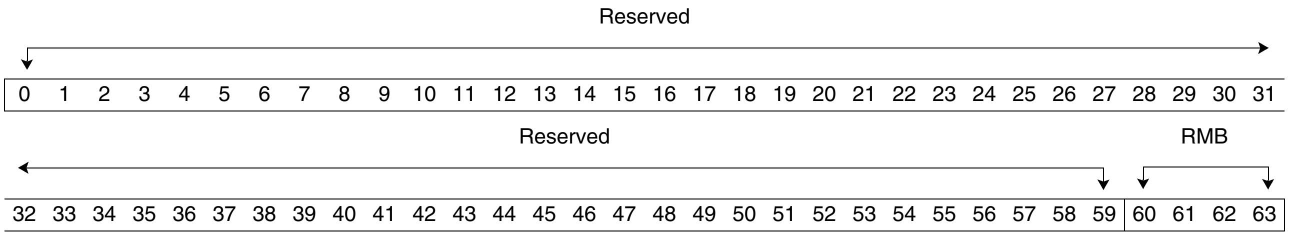
The MFC Real-Mode Address Boundary Register is an implementation-dependent register. Access to this register should be privileged.

Figure 14-2. Real-Mode Storage Boundary (showing all DMA transfers)



Access Type Read/Write

Base Address Offset (BP_Base | P1(n)) + x‘0900’, where n is an SPE number.



Bits	Field Name	Description
0:59	Reserved	Set to zeros.
60:63	RMB	Real-mode boundary. If this field is set to a value of ‘n’, only those accesses within the first 2^{n+27} bytes for $1 \leq n \leq 15$ of the real address space are considered well behaved and cacheable in real addressing mode (MFC_SR1[R] = ‘0’). All accesses outside the first 2^{n+27} bytes of the real address space are neither well behaved nor cacheable in real addressing mode. If the RMB field is set to a value of x‘0’, no accesses are considered well behaved and cacheable in real addressing mode.

15. MFC Privileged Facilities

The registers in these sections are only accessed by software. The memory flow controller (MFC) privilege 1 facilities include the following registers:

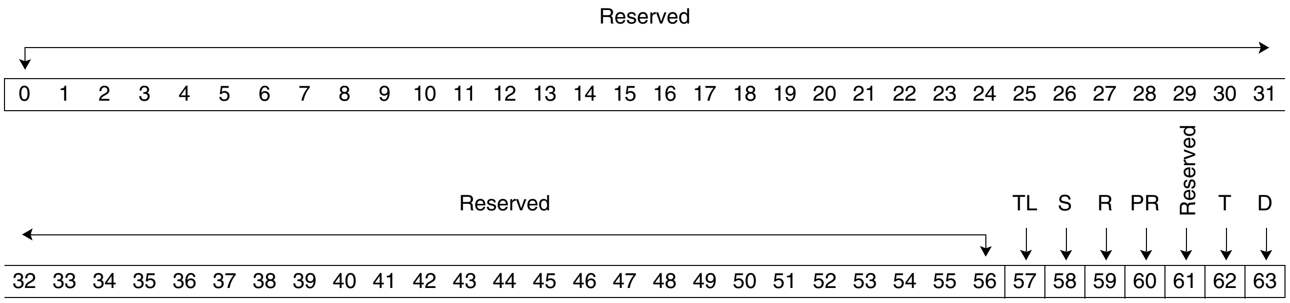
- MFC State Register One
- MFC Logical Partition ID Register (see page 223)
- MFC Storage Description Register (see page 224)
- MFC Data Address Register (see page 225)
- MFC Data Storage Interrupt Status Register (see page 226)
- MFC Address Compare Control Register (see page 227)
- MFC Local Storage Address Compare Facility (see page 229)
- MFC Command Error Register (see page 231)
- MFC Data Storage Interrupt Pointer Register (see page 232)
- MFC Control Register (see page 233)
- MFC Atomic Flush Register (see page 236)
- SPU Outbound Interrupt Mailbox Register (see page 237)

15.1 MFC State Register One (MFC_SR1)

MFC State Register One contains configuration information that is controlled by a hypervisor. Access to this register should be privileged.

Access Type Read/Write

Base Address Offset (BP_Base | P1(n)) + x‘0000’; where n is an SPE number.



Bits	Field Name	Description
0:56	Reserved	Reserved.
57	TL	Software or hardware page table search 0 Translation lookaside buffer (TLB) misses are handled by the hardware (segment lookaside buffer [SLB] and page table entry [PTE] misses are always handled by software). 1 TLB misses are handled by software.

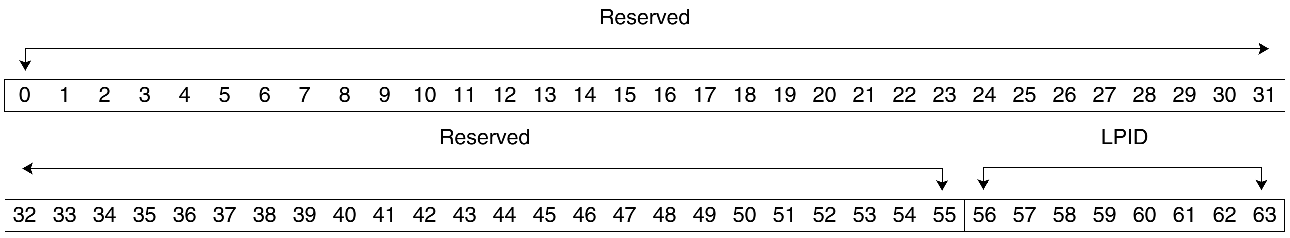
Bits	Field Name	Description
58	S	Synergistic processor unit (SPU) master run control. Setting this bit to zero suspends the dispatch of instructions in the SPU. The state of this bit does not affect the state of the SPU Run Status in the SPU Status Register (see page 97). 0 SPU is stopped. 1 SPU is controlled by the SPU Run Control Register (see page 96). **Programming Note:** The SPU Master Run Control (S) bit allows privileged software to keep an SPU from running even if an application set the SPU Run bit in the SPU Run Control Register (see page 96). This bit is useful during power savings mode.
59	R	Relocate. 0 MFC translation is off. 1 MFC translation is on. **Programming Note:** The MFC Relocate (R) bit controls how effective addresses in MFC commands are translated into real addresses. If the relocate control specifies translation off (R = ‘0’), the effective addresses used in MFC commands are real addresses and are subject to the real-mode offset control facility. If the relocate control is enabled (R = ‘1’), the effective addresses used in MFC commands are translated. The SLB, TLB, and page table facilities are used to translate the effective address to a virtual address, and then to a real address. With translation on, the real-mode offset control is not involved.
60	PR	Problem state. 0 The MFC has privileged-state access to pages. 1 The MFC has problem-state access to pages. **Programming Note:** The MFC Problem-State (PR) bit is set by privileged software based on the use of the associated SPU. If the SPU is to be a privileged software resource (not under direct control of an application) and the function requires the SPU to issue MFC commands with privileged-state access to pages, this bit should be cleared. If the SPU function is controlled by an application, its MFC access should be restricted to problem state by the PR bit. The Problem State Control bit interacts with the Ks and Kp Storage Key bits in the SLB in combination with the Page Protection (PP) bits in the page table, as defined in *PowerPC Architecture, Book III*. The problem-state control is only effective in MFC translation on state (R = ‘1’).
61	Reserved	Reserved.
62	T	Bus **tlbie** enable. 0 Ignore **tlbie** commands on the bus. 1 Invalidate TLB entries in response to a **tlbie** on the bus. **Programming Note:** The **tlbie** (T) bit is typically enabled if page tables are used by multiple PowerPC Processor Elements (PPEs) and MFCs. If both **tlbie**-managed and MFC managed TLBs are enabled, the MFC will participate in broadcast **tlbie** operations (initiated by a PPE **tlbie** instruction). If disabled, the MFC will ignore the **tlbie** broadcast. This state is useful when each MFC uses its own private page table.
63	D	Local storage real address decode. 0 Disable system real address of local storage. The MFC does not decode the local storage area for the associated SPU. All accesses are ignored. This can be done to save power 1 Enable system real address of local storage The MFC decodes the local storage area for the associated SPU. All accesses are performed. **Programming Note:** Privileged software can use the Local Storage Real Address Decode Enable/Disable (D) bit to enable an application to physically address local storage. In addition to setting the D bit, privileged software must set up the page table or an implementation-dependent I/O translation mechanism to map the local storage into the application’s effective address space. Setting this bit does not affect the addressing of local storage by SPU load and store instructions or MFC commands. This bit only affects the access of the local storage using an effective address.

15.2 MFC Logical Partition ID Register (MFC_LPID)

The PowerPC Architecture provides a logical partitioning (LPAR) facility to permit processors and portions of main storage to be allocated to logical groups or partitions. For more information about the LPAR, see *PowerPC Architecture, Book III*. The Cell Broadband Engine Architecture (CBEA) extends the LPAR facility to allow SPEs to also be assigned to partitions. The MFC Logical Partition ID Register contains a value that identifies the partition to which an SPE is assigned.

Access Type Read/Write

Base Address Offset (BP_Base | P1(n)) + x‘0008’; where n is an SPE number.



Bits	Field Name	Description
0:55	Reserved	Reserved
56:63	LPID	Logical partition ID.

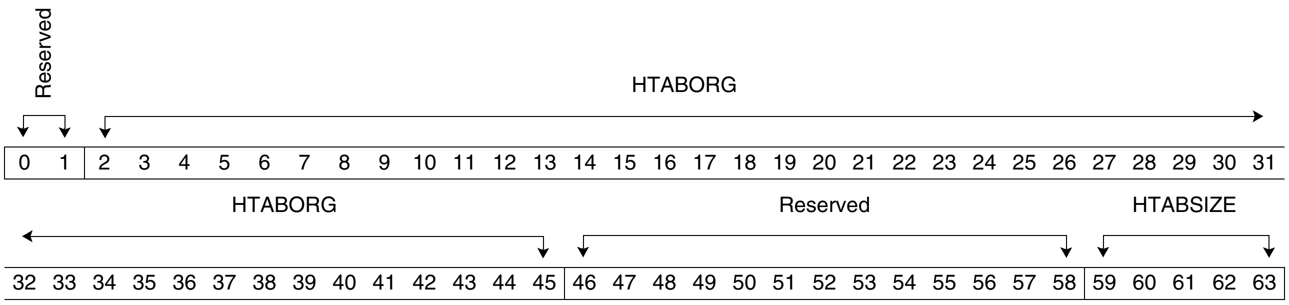
15.3 MFC Storage Description Register (MFC_SDR)

The MFC Storage Description Register contains the starting address in main storage of the page table for the associated MFC and the size of the page table. The MFC Storage Description Register provides the same function as the PowerPC Storage Description Register (SDR1). For more information about the SDR1, see *PowerPC Architecture, Book III*.

When an SPE is configured for software TLB management (that is, MFC_SDR[TL] = '1'), the value in this register is not used.

Access Type Read/Write

Base Address Offset (BP_Base I P1(n)) + x‘0400’; where n is an SPE number.



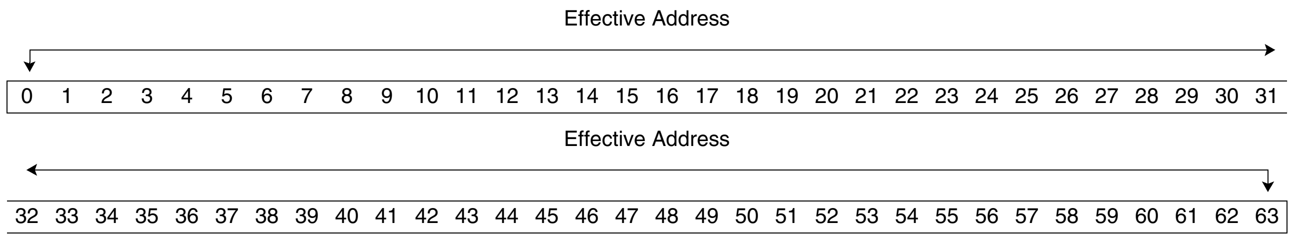
Bits	Field Name	Description
0:1	Reserved	Reserved.
2:45	HTABORG	Page table origin (real address of the page table). The HTABORG field in MFC_SDR contains the high-order 44 bits of the 62-bit real address of the page table. The page table is thus constrained to lie on a 2^{18} byte (256 KB) boundary at a minimum. The number of low-order zero bits in HTABORG must be greater than or equal to the value in HTABSIZE. For implementations that support a real address size of only m bits, where m is less than 62, the upper bits of the page table origin are treated as reserved bits. Software must set them to zeros.
46:58	Reserved	
59:63	HTABSIZE	Encoded size of page table. The HTABSIZE field in MFC_SDR contains an integer giving the number of bits (in addition to the minimum of 11 bits) from the hash that are used in the page table index. This number must not exceed 28.

15.4 MFC Data Address Register (MFC_DAR)

The MFC Data Address Register contains the effective address associated with an MFC data segment interrupt or an MFC data storage interrupt. The function of this register is similar to the PowerPC Data Address Register (DAR). For more information about the DAR, see *PowerPC Architecture, Book III*. Access to this register should be privileged.

Access Type Read/Write

Base Address Offset (BP_Base | P1(n)) + x‘0620’; where n is an SPE number.



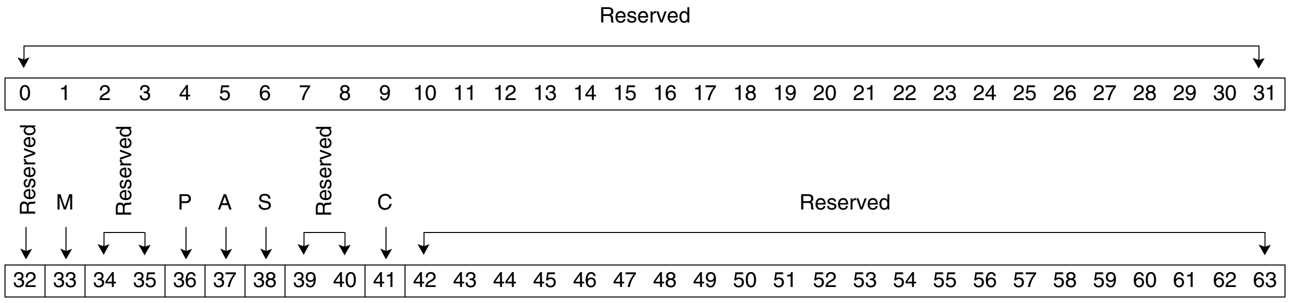
Bits	Field Name	Description
0:63	Effective Address	Effective address associated with the data segment or data storage interrupt.

15.5 MFC Data Storage Interrupt Status Register (MFC_DSISR)

The MFC Data Storage Interrupt Status Register contains the status that defines the cause of the MFC data storage interrupt. The function of this register is similar to the PowerPC Data Storage Interrupt Status Register (DSISR). For more information, see *PowerPC Architecture, Book III*. Access to this register should be privileged.

Access Type Read/Write

Base Address Offset (BP_Base | P1(n)) + x‘0610’; where n is an SPE number.



Bits	Field Name	Description
0:31	Reserved	Set to zeros.
32	Reserved	Set to ‘0’.
33	M	Set to ‘1’ if the PTE is not found or a TLB miss occurs while in software-managed TLB mode.
34:35	Reserved	Set to zeros.
36	P	Set to ‘1’ if the access is not permitted by the storage protection mechanism.
37	A	Set to ‘1’ if the atomic access is to a write through or caching inhibited page.
38	S	Set to ‘1’ if the access was a **put*[rlfbs]***, a **putll*[u]*c**, or an **sdcrz** operation.
39:40	Reserved	Set to zeros.
41	C	Set to ‘1’ if a data address compare match occurs (all DMA issue activity is halted).
42:63	Reserved	Set to zeros.

15.6 MFC Address Compare Control Register (MFC_ACCR)

The MFC data address compare mechanism allows the detection of DMA access to a virtual page marked with the Address Compare (AC) bit in the page table entry (PTE) set and to a range within the local storage. This facility is normally used for debug. This debug mechanism is controlled by the MFC Address Compare Control Register for both effective address and local storage address compares. For effective address compares, the AC bit in the page table entry controls which pages are included in the comparison. For local storage compares, the MFC local storage address compare facility contains the address used for the comparison and a mask indicating which address bits should be included in the comparison.

An effective address compare match occurs for a **put**, **get**, **getllar**, **putll[u]c**, or **sdcrz** operation if the following conditions are true for any byte accessed (including effective addresses used in list commands):

- The PTE AC bit is set, the operation is a **put**, and the MFC_ACCR Put (Ep) bit is set.
- The PTE AC bit is set, the operation is a **get**, and the MFC_ACCR Get (Eg) bit is set.

A local storage address compare match occurs for a **put**, **get**, **getllar**, **putllc**, and **putlluc** operation if, for any byte access, the following conditions are true (including local storage addresses accessed by list commands). The local storage address value compare occurs before the SPU Local Storage Limit Register wrap (if any) is applied.

- The operation is a **put**, the local storage address being read matches the address range specified by the bit-wise AND of the local storage compare address mask and the local storage compare address in the MFC local storage address compare facility, and the Local Storage Put bit (Lp) in the MFC_ACCR is set.
- The operation is a **get**, the local storage address being written matches the address range specified by the bit-wise AND of the local storage compare address mask and the local storage compare address in the MFC local storage address compare facility, and the Local Storage Get bit (Lg) in the MFC_ACCR is set.

Once an effective address compare match occurs, an MFC data storage interrupt is presented, and the C bit is set in the MFC Data Storage Interrupt Status Register (see page 226). Once a local storage address compare match occurs, a class 1 interrupt is presented, and the LP or LG bits are set in the Class 1 Interrupt Status Register (see page 281). For both effective address and local storage address compares, all MFC DMA operations are stopped at the command that caused the compare match. Some portion or all of the command might have been completed before the stop. Due to the weakly ordered model, additional commands might also have been started when the address compare occurs. For effective address compares, the MFC Data Storage Interrupt Pointer Register (see page 232) will contain the index of the command that triggered the address compare. The MFC_DAR will contain the effective address of the access that triggered the compare condition.

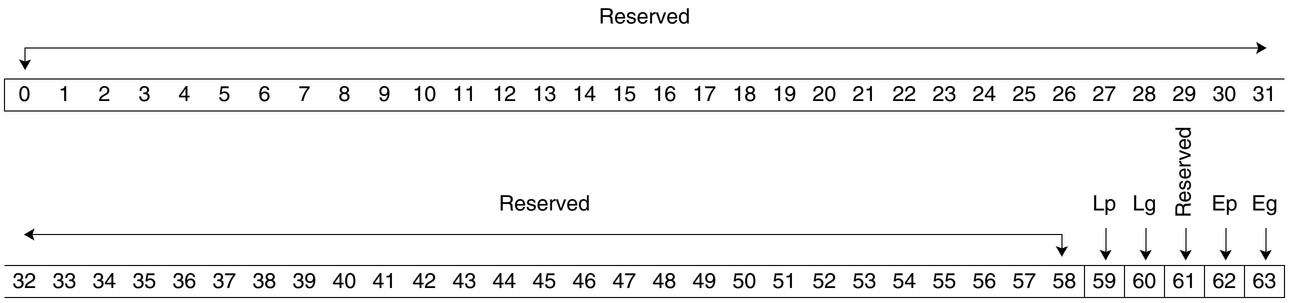
For local storage address compare matches, the MFC_LSCRR contains the local storage address of the access that triggered the compare condition and the index of the command that triggered the address compare.

For both effective address and local storage address compare stops, the DMA operations can be resumed by writing the MFC Control Register with the Sc bit cleared, which indicates resume normal operation. In addition to setting the Sc bit to zero, the AC bit in the PTE must be reset. The R bit in the MFC Control Register must also be set to resume a command stopped due to an effective address compare. Setting the R bit causes the MFC to retranslate the command.

Access to this register should be privileged.

Access Type Read/Write

Base Address Offset (BP_Base | P1(n)) + x‘0600’; where n is an SPE number.



Bits	Field Name	Description
0:58	Reserved	Reserved.
59	Lp	Local storage put. Enable of local storage address for **put** commands (includes **putll[u]c** operations).
60	Lg	Local storage get. Enable of local storage address for **get** commands.
61	Reserved	Reserved.
62	Ep	Effective address put. Enable of effective address for **put** commands (includes **sdcrz** and **putll[u]c** operations).
63	Eg	Effective address get. Enable of effective address for **get** commands.
Note: There is no restriction on the setting of enables in this register.		

15.7 MFC Local Storage Address Compare Facility

Section 15.6 MFC Address Compare Control Register on page 227 explains how to use this facility. This facility consists of the following registers:

- MFC Local Storage Address Compare Register
- MFC Local Storage Compare Result Register (see page 230)

15.7.1 MFC Local Storage Address Compare Register (MFC_LSACR)

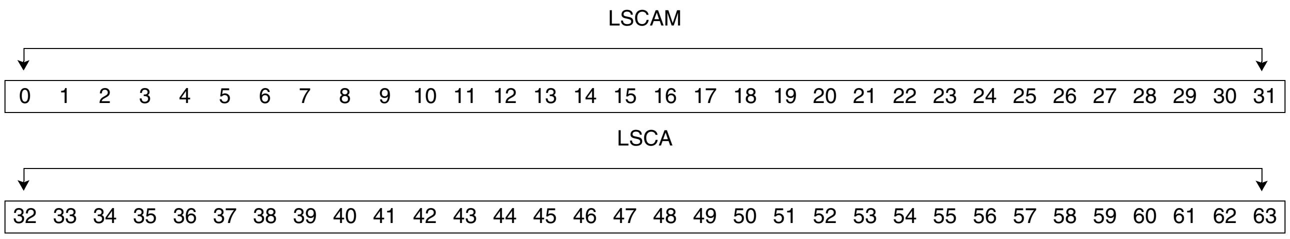
The MFC local storage address compare facility contains the local storage address and local storage address mask to be used in the MFC local storage address compare operation selected by the MFC Address Compare Control Register (see page 227). Access to this register should be privileged.

A local storage address compare occurs when the local storage address (LSA) accessed is within the range of addresses specified by the bit-wise AND of the Local Storage Compare Address Mask (LSCAM) and the Local Storage Compare Address (LSCA) fields.

- Get Match = ((LSCA & LSCAM) == (LSA & LSCAM)) && MFC_ACCR[Lg]
- Put Match = ((LSCA & LSCAM) == (LSA & LSCAM)) && MFC_ACCR[Lp]

Access Type Read/Write

Base Address Offset (BP_Base | P1(n)) + x‘0808’; where n is an SPE number.



Bits	Field Name	Description
0:31	LSCAM	Local storage compare address mask. The number of bits implemented in the mask field is implementation dependent. A ‘0’ value written to a bit in this field makes the corresponding bit in the local storage address a “don’t care” in the MFC local storage address compare operation. A ‘1’ value written to a bit in this field includes the corresponding local storage address bit in the MFC local storage address compare operation.
32:63	LSCA	Local storage compare address. The number of upper bits implemented in the address and mask fields is implementation dependent.

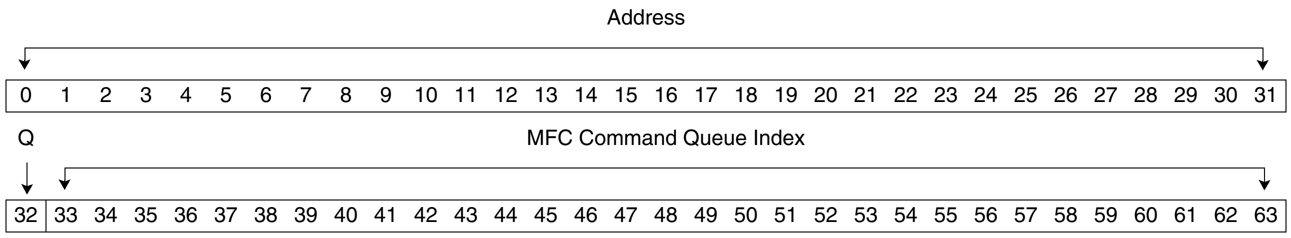
15.7.2 MFC Local Storage Compare Result Register (MFC_LSCRR)

The MFC Local Storage Compare Result Register contains the local storage address that triggered the compare along with the MFC command queue index of the MFC command that triggered the compare stop. The contents of this register are only valid when a class 1 interrupt occurs with the Lp or Lg Interrupt Status bits set. (For more information about class 1 interrupts, see *Section 21.5.2* on page 272.) The Q bit indicates if the command was executed from the MFC proxy command queue or MFC SPU command queue.

Access to this register should be privileged. The contents of this register become indeterminate once MFC operation is resumed.

Access Type Read Only

Base Address Offset (BP_Base | P1(n)) + x‘0810’; where n is an SPE number.



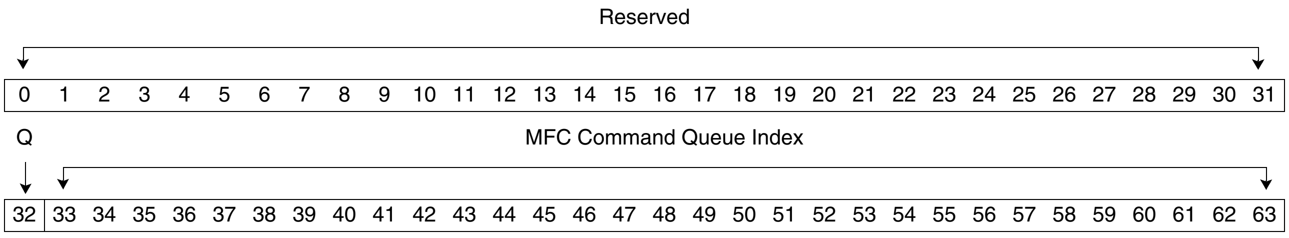
Bits	Field Name	Description
0:31	Address	Address of access triggering local storage address compare
32	Q	Command queue 0 Index is for the MFC proxy queue. 1 Index is for the MFC SPU queue.
33:63	MFC Command Queue Index	MFC command queue index. Points to the queue entry that triggered the compare stop.

15.8 MFC Command Error Register (MFC_CER)

The MFC Command Error Register contains the index of the command in the MFC command queue associated with an MFC error condition. When the MFC detects an error, all processing is suspended until the error is cleared and the MFC DMA operation is restarted. The MFC DMA operation is restarted by writing the MFC Control Register (see page 233) with the MFC_CNTL[R] bit set.

Note: Command errors can occur on the MFC proxy and MFC SPU command queues. The MFC must stop execution on the first error. The MFC Command Error Register must point to the command that caused the first error.

Access Type	Read Only
Base Address Offset	(BP_Base I P1(n)) + x‘0C00’; where n is an SPE number.



Bits	Field Name	Description
0:31	Reserved	Set to zeros.
32	Q	Command queue. 0 Index is for the MFC proxy queue. 1 Index is for the MFC SPU queue.
33:63	MFC Command Queue Index	MFC command queue index. Points to the queue entry that caused the command error. The number of bits implemented in this field is implementation dependent.

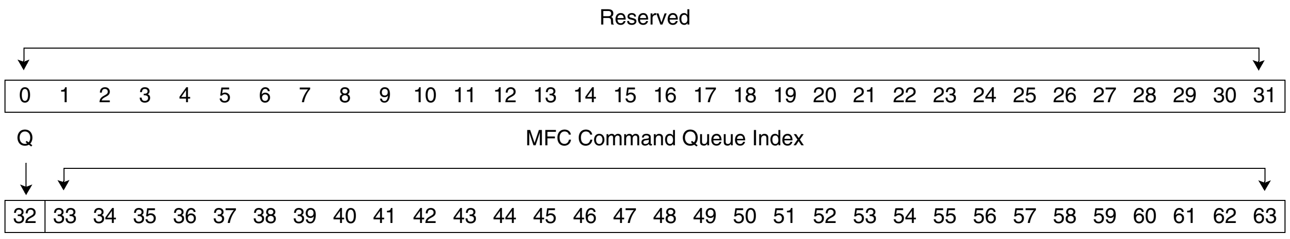
15.9 MFC Data Storage Interrupt Pointer Register (MFC_DSIPR)

The MFC Data Storage Interrupt Pointer Register contains the index for the command in the DMA command queue associated with 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 Data Storage Interrupt Status Register (see page 226).

Access Type Read Only

Base Address Offset (BP_Base | P1(n)) + x‘0800’; where n is an SPE number.



Bits	Field Name	Description
0:31	Reserved	Set to zeros.
32	Q	Command queue. 0 Index is for the MFC proxy queue. 1 Index is for the MFC SPU queue.
33:63	MFC Command Queue Index	MFC command queue index. Points to the command that caused the data storage interrupt. The number of bits implemented in this field is implementation dependent.

Implementation Note:

Only one translation fault can be outstanding. The implementation can either stop all MFC command queue processing on the first translation error, or it can continue processing. If processing continues, all ordering rules must be followed. (A command must not be processed if it is dependent on a command that is waiting for a translation fault to be resolved.) The state of the MFC must appear as if the command (or partial command) was never issued. This is also the case if a second translation fault occurs.

15.10 MFC Control Register (MFC_CNTL)

The MFC Control Register allows privileged software to govern the operation of the MFC and the MFC commands. There is one register for each SPU within an SPU group.

Setting the Suspend Control (Sc) bit while the Suspend Mask (Sm) bit is a ‘0’ causes the MFC to stop executing commands. The MFC_CNTL[Sc] bit is ignored if MFC_CNTL[Sm] is a ‘1’ when the MFC Control Register is written. MFC commands might still be enqueued while an MFC command queue is suspended. To change the state of MFC_CNTL[Sc], MFC_CNTL[Sm] must be set to zero. If MFC_CNTL[Sm] is set to one, the state of MFC_CNTL[Sc] will not be updated. MFC_CNTL[Sm] should be written as a ‘1’ value when the MFC Control Register is written with no intent to change the current MFC operation state (suspended or normal).

Setting the Purge (Pc) bit in this register causes the MFC to remove all commands from the MFC command queue. Hardware resets this bit when a purge operation completes. MFC commands are not enqueued while the MFC command queue is in the purge state.

Setting the Restart (R) bit of this register causes the MFC command with a pending translation fault to be reissued. Hardware resets MFC_CNTL[R] automatically after a pending MFC command is reissued.

Note: MFC_CNTL[R] should not be set to resume an MFC command that stopped due to a local storage address compare.

The restart operation is only effective if the MFC command queue is in normal queue operational status. Software must set MFC_CNTL[R] to ‘1’ to resume an MFC command either after one of the faults listed below or after a page protection fault has been indicated.

- A fault is either a data segment interrupt or a data storage interrupt with the Miss (M) bit set in the MFC Data Storage Interrupt Status Register (see page 226).
- A page protection fault is a data storage interrupt with the Protection (P) bit set in the MFC_DSISR.

Software can set the MFC_CNTL[Sc] bit and the MFC_CNTL[R] bit in the same memory-mapped I/O (MMIO) write operation.

The Decrementer Halt (Dh) bit allows privileged software to stop the decrementer. The decrementer remains halted until MFC_CNTL[Dh] is reset and the SPU issues a write channel (**wrch**) instruction to the SPU decrementer.

The Decrementer Status (Ds) bit reflects the state of the decrementer (running or not running). This allows privileged software to determine if the application running in the SPU used the decrementer. The state of the decrementer is required for the context save and resume of an SPE.

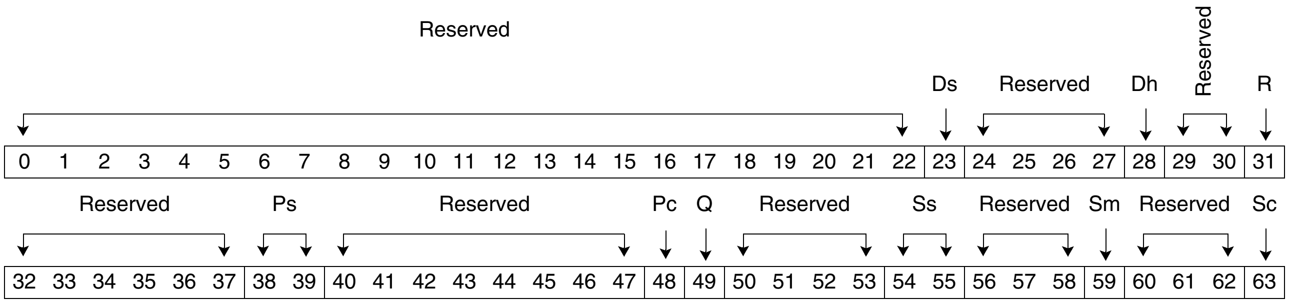
When either an MFC command error, an effective address compare stop, a local storage address compare, or an MFC hardware error occurs, hardware sets MFC_CNTL[Sc] to suspend MFC command queue operation. MFC operations are suspended for both MFC command queues. The MFC command queue status changes to MFC command queue operation suspended (Ss equals ‘11’) when all outstanding DMA transfers are complete.

To resume normal MFC command queue operations:

- After an effective address compare stop, the AC bit in the PTE must be reset. Then, the MFC Control Register must be written to set MFC_CNTL[Sc] to ‘0’ and to reset MFC_CNTL[R] to ‘1’. Resetting MFC_CNTL[R] causes the MFC to retranslate the command.
- After a local storage address compare stop, the MFC Control Register must be written to set MFC_CNTL[Sc] to ‘0’. Do not set the MFC_CNTL[R] bit to resume an MFC command stopped due to a local storage address compare.
- After an MFC command or an MFC hardware error, the Purge Sequence bits (Pc and Ps) should be issued before setting normal MFC command queue operation mode.

Access Type Read/Write

Base Address Offset (BP_Base | P2(n)) + x‘03000’; where n is an SPE number.



Bits	Field Name	Description
0:22	Reserved	Reserved.
23	Ds	SPU decrementer status. This is a read only field. Data written to this field will be ignored. 0 Decrementer not running. 1 Decrementer running.
24:27	Reserved	Reserved.
28	Dh	SPU decrementer halt. 0 Decrementer is allowed to run if activated by a write to the SPU Write Decrementer Channel. 1 Decrementer halted.
29:30	Reserved	Reserved.
31	R	Restarts the MFC command that caused the translation fault or an effective address compare stop. **Note:** This bit must *not* be set to resume an MFC command stopped due to a local storage address compare. This bit must be set to ‘1’ to restart the MFC command operation that caused a translation fault or an effective address compare stop. This bit is automatically reset by hardware after the MFC command has been resumed. 0 No MFC command restart requested. 1 Write: Restart MFC command. Read: MFC command reissue pending.
32:37	Reserved	Reserved.
38:39	Ps	MFC command queue purge status. This is a read only field. Data written to this field will be ignored. 00 Purge request not outstanding. 01 Purge of MFC command queues in process (some implementations can choose not to implement this state. 11 Purge of MFC command queues is complete.

Bits	Field Name	Description
40:47	Reserved	Reserved.
48	Pc	Purge MFC commands from the MFC SPU command queue and MFC proxy command queue. 0 No purge of MFC command queues requested. 1 Purge MFC command queues request.
49	Q	MFC command queue empty status. This is a read only field. Data written to this field is ignored. 0 MFC SPU command queue and MFC proxy command queue are not empty. 1 MFC SPU command queue and MFC proxy command queue are both empty. All MFC commands are complete.
50:53	Reserved	Reserved.
54:55	Ss	MFC command queues suspend status. This is a read only field. Data written to this field will be ignored. 00 Normal MFC command queues operation. 01 Suspend of MFC command queues in process (some implementations can choose not to implement this state). 11 MFC command queue operation suspended (both queues).
56:58	Reserved	Reserved.
59	Sm	Suspend mask This bit is used to control the effect of the MFC_CNTL[Sc] bit when writing the MFC Control Register. System software uses this bit to avoid unintentional changes to the MFC state. 0 Effect of Sc bit is enabled. 1 Effect of Sc bit is disabled.
60:62	Reserved	Reserved.
63	Sc	Suspend control. Suspend MFC command queues operation. 0 Normal MFC command queue operation request (both queues). 1 Suspend MFC command queue operation request (both queues).

Implementation Note:

Once the MFC_CNTL[Sc] bit is set, hardware must stop issuing any new transactions. It must record the state of the MFC command so that it can be restarted in the future. However, hardware can complete any current outstanding transactions. If an MFC command is being divided into a series of smaller transactions, hardware must stop the process. Hardware is allowed to complete any transactions divided into a series of smaller transactions before MFC_CNTL[Sc] was set. All MFC commands, including any partial transactions, must be flushed before setting the suspended status. Hardware must purge all commands (partial or complete) once the Purge bit is set, but it can complete any current outstanding transactions.

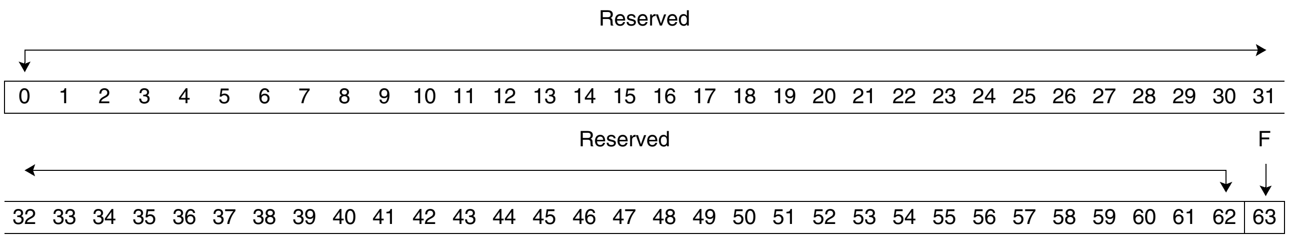
15.11 MFC Atomic Flush Register (MFC_Atomic_Flush)

The MFC Atomic Flush Register is implementation dependent, and access should be privileged. Privileged software uses this register to clear the contents of the cache used for atomic DMA commands and releases any current reservations. Data in the cache that is considered modified is pushed to memory, and the line is invalidated. Valid lines in the cache that are not considered modified are invalidated.

For this operation to work properly, privileged software must suspend the MFC command queues.

Access Type Read/Write

Base Address Offset (BP_Base | P1(n)) + x‘0200’; where n is an SPE number.



Bits	Field Name	Description
0:62	Reserved	Reserved.
63	F	Flush enable or status. When software sets this bit to ‘1’, the contents of the atomic unit are flushed. Hardware resets the bit when the flush operation is complete. Software should never write a ‘0’ to this field. Software should poll this register for completion of the flush operation.

15.12 SPU Outbound Interrupt Mailbox Register (SPU_Out_Intr_Mbox)

The SPU Outbound Interrupt Mailbox Register is used to read 32 bits of data from the SPU outbound interrupt mailbox queue of the corresponding SPU. The SPU Outbound Interrupt Mailbox Register has a corresponding channel (see page 140) for writing data into the SPU outbound interrupt mailbox queue.

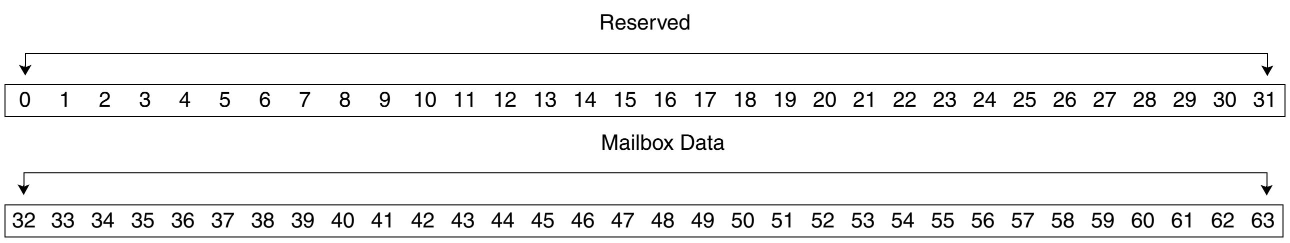
A write channel (**wrch**) instruction that targets the SPU outbound interrupt mailbox queue loads the 32 bits of data specified in the instruction into a mailbox queue for other processors or devices to read. An MMIO read from this register always returns the data in the order written by the SPU. The data returned on a read of an empty SPU outbound interrupt mailbox queue is undefined.

An MMIO read of the SPU Mailbox Status Register (see page 104) returns the status of the mailbox queues. The number of valid queue entries in the SPU Outbound Interrupt Mailbox Register is given in the SPU_Out_Intr_Mbox_Count field of the SPU Mailbox Status Register (that is, SPU_Mbox_Stat [SPU_Out_Intr_Mbox_Count]).

An MMIO read of the SPU Outbound Interrupt Mailbox Register sets a pending SPU outbound interrupt mailbox available event. If the amount of data remaining in the mailbox queue is below an implementation-dependent threshold, and if this condition is enabled (that is, SPU_WrEventMask[Me] set to ‘1’), the SPU Read Event Status Channel is updated (that is, SPU_RdEventStat[Me] is set to ‘1’) and its channel count is set to 1. This causes an SPU outbound interrupt mailbox available event.

Access Type MMIO: Read

Base Address Offset (BP_Base | P2(n)) + x‘04000’; where n is an SPE number.



Bits	Field Name	Description
0:31	Reserved	Set to zeros.
32:63	Mailbox Data	Application-specific mailbox data. Each application can uniquely define the mailbox data.

Implementation Note:

The MFC must not acknowledge the write of the SPU Outbound Interrupt Mailbox Register until a processor or other device has read the contents of the mailbox.

16. SPU Privileged Facilities

Synergistic processor unit (SPU) privileged facilities include these registers:

- SPU Privileged Control Register
- SPU Local Storage Limit Register (see page 241)
- SPU Channel Access Facility, which includes:
  - SPU Channel Index Register (see page 242)
  - SPU Channel Data Register (see page 243)
  - SPU Channel Count Register (see page 244)
- SPU Configuration Register (see page 245)

16.1 SPU Privileged Control Register (SPU_PrivCntl)

The SPU Privileged Control Register lets privileged software control the execution environment of the SPU. The SPU Privileged Control Register can be used to place the SPU into single-instruction-step mode or to generate a privileged attention event.

Single-instruction-step mode remains in effect until cancelled by writing this register with the Single-Step-Mode bit reset. When single-step mode is active and the SPU is started using the SPU Run Control Register (see page 96) or the SPU start command on a direct memory access (DMA) operation, a single instruction or instruction group is executed. The SPU is stopped, and a class 2 SPU halted interrupt (if enabled) is presented to the PowerPC Processor Element (PPE). The stopped-by-single-step indicator is set in the SPU Status Register. Other stop conditions can also be reported along with a single-step stop. Single-step operation is not available when the SPU is operating in isolation mode (the Isolate [IS] bit in the SPU Status Register = ‘1’). Setting the single-step mode is ignored if the SPU is in an isolated state.

When this register is written with the Attention Event Request bit set, a privileged attention event is raised on the SPU. SPU acknowledgment of this event resets this condition. When reading from this register, the current state of single-step mode is provided. However, the Attention Event Request bit always returns ‘0’.

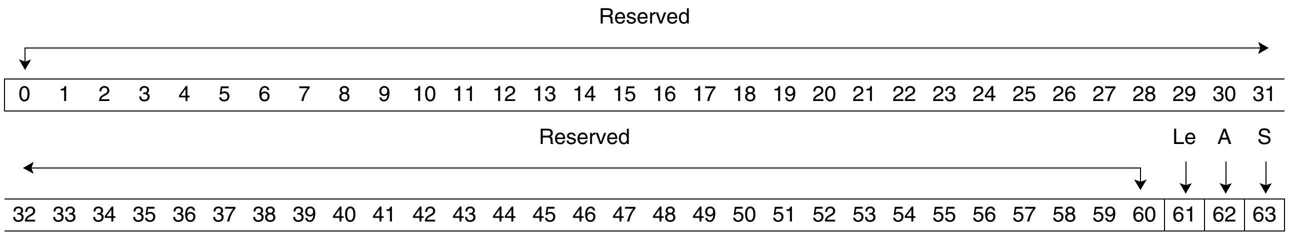
Privileged code can use the privileged attention event mechanism to trigger an SPU event when the SPU software supports the SPU privileged attention event (the privileged attention event is enabled). When the SPU software supports the SPU privileged attention event, it can support requests specific to the operating environment such as light-weight, application-assisted context switching of SPUs.

Privileged code can use the load enable to prevent an application from issuing an isolation load request to put the SPU into an isolated state.

Access to this register should be privileged.

Access Type Read/Write

Base Address Offset (BP_Base | P2(n)) + x‘04040’; where n is an SPE number.



Bits	Field Name	Description
0:60	Reserved	Set to zeros.
61	Le	Isolation load request enable. 0 PPE or SPU isolate load request ignored. 1 PPE or SPU isolate load request allowed. Writing ‘11’ to the SPU Run Control Register (see page 96) causes the SPU to transition into the isolated load state.
62	A	Attention event request. 0 No SPU privileged attention event requested. 1 SPU privileged attention event requested. **Note:** The Attention Event Request bit always returns ‘0’.
63	S	Single-step mode. 0 Normal operation. 1 SPU will issue an instruction or set of instructions and then stop.

16.2 SPU Local Storage Limit Register (SPU_LSLR)

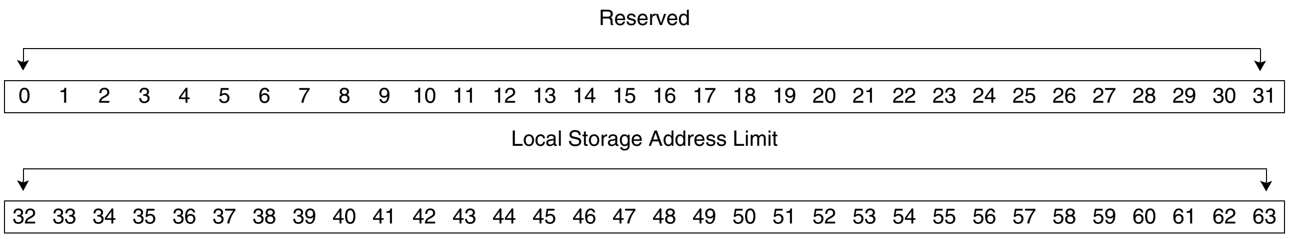
The SPU Local Storage Limit Register allows the size of local storage available to an application to be artificially limited. This register enables privileged software to provide backwards compatibility for applications that are sensitive to the size of local storage. If an application performs a quadword load or store from the SPU that is beyond the range of the SPU Local Storage Limit Register, the operation occurs at the wrapped address.

When an isolation load is requested, the contents of the SPU_LSLR are forced to the maximum value for the implementation. The contents of the SPU_LSLR are not restored to the previous value when exiting an isolated state.

Access to this register should be privileged.

Access Type Read/Write

Base Address Offset (BP_Base | P2(n)) + x‘04058’; where n is an SPE number.



Bits	Field Name	Description
0:31	Reserved	Set to zeros.
32:63	Local Storage Address Limit	Implementation dependent.

16.3 SPU Channel Access Facility

The SPU channel access facility initializes, saves, and restores the SPU channels. The facility consists of three memory-mapped I/O (MMIO) registers: the SPU Channel Index Register, the SPU Channel Count Register, and the SPU Channel Data Register. The SPU Channel Index Register is a pointer to the channel whose count is accessed by the SPU Channel Count Register and whose data is accessed by the SPU Channel Data Register.

Note: There is also an internal Pending Event Register that is described in *Section 9.11 SPU Event Facility* beginning on page 150.

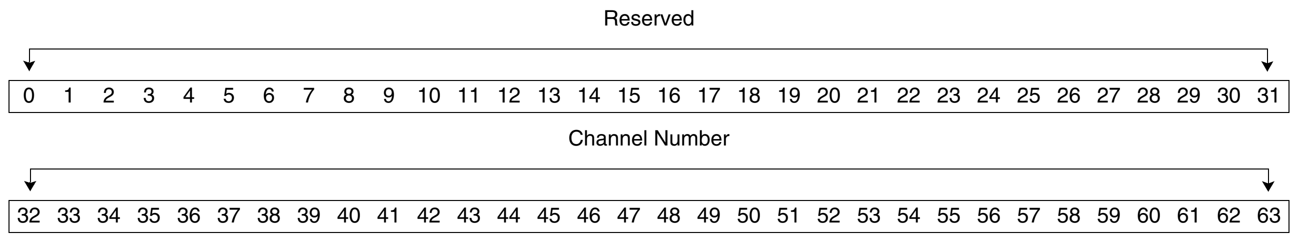
16.3.1 SPU Channel Index Register (SPU_ChnlIndex)

The SPU Channel Index Register selects which SPU channel is accessed using the SPU Channel Count Register or the SPU Channel Data Register.

Access to this register should be privileged.

Access Type Write[1]

Base Address Offset (BP_Base | P2(n)) + x'04060'; where n is an SPE number.



Bits	Field Name	Description
0:31	Reserved	Set to zeros.
32:63	Channel Number	This field contains the channel number that is modified by the SPU Channel Count Register or the SPU Channel Data Register. The number of bits implemented for this field is implementation dependent. See the specific implementation documentation for more information.

1. Read should be supported for diagnostic purposes.

16.3.2 SPU Channel Data Register (SPU_ChnlData)

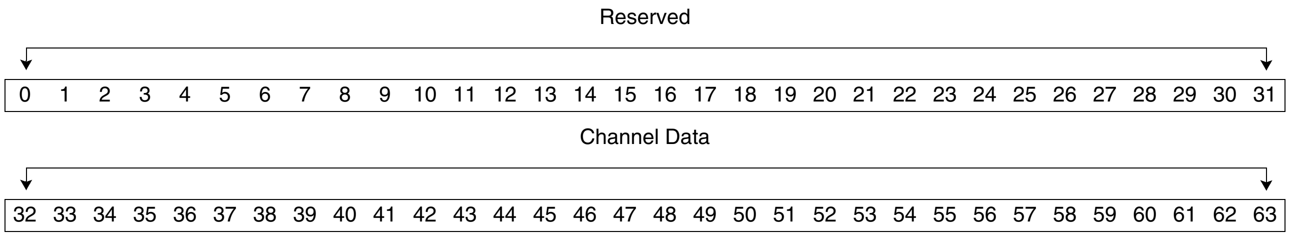
The SPU Channel Data Register is used to read or initialize the SPU channel data selected by the SPU Channel Index Register (see page 242). Initializing or restoring channel data with this register has no effect on the channel count associated with the channel. In addition, it does not generate any channel packet activity on the channel interface.

When the channel being serviced by the channel data and index ports supports more than 1-deep first-in, first-out queues (FIFOs), writing the channel index to specify the channel selects the oldest FIFO entry to be accessed by the channel data port. Successive accesses to the channel data port will then access successively newer entries in the FIFO. The behavior of accesses beyond the depth of the FIFO is implementation dependent; such accesses should be avoided.

Reading or writing the SPU read event status data through this interface provides direct access to the internal Pending Event Register. Therefore, the external event mask has no effect on the data read from this channel when using the SPU channel access facility.

Access Type Read/Write

Base Address Offset (BP_Base | P2(n)) + x‘04070’; where n is an SPE number.



Bits	Field Name	Description
0:31	Reserved	Set to zeros.
32:63	Channel Data	This field is used to initialize the data for the channel identified by the SPU Channel Index Register. Access to this resource should be privileged. The number of bits implemented for this register is channel specific. Only a subset of the implemented channels can be accessed through this register. See the specific implementation documentation for more information.

Programming Note:

To support SPE context save, restore, and debug; the following channels must be supported through this interface:

- SPU Pending Event Register (supporting SPU Read Event Status Channel x‘0’)
- SPU Write Event Mask Channel (x‘1’)
- SPU Signal Notification 1 Channel (x‘3’)
- SPU Signal Notification 2 Channel (x‘4’)
- MFC Read Tag-Group Status Channel (x‘18’)
- MFC Read List Stall-and-Notify Tag Status Channel (x‘19’)
- MFC Read Atomic Command Status Channel (x‘1B’)
- SPU Read Inbound Mailbox Channel (x‘1D)

Channel data for channels x‘0’, x‘1’, x‘3’, x‘4’, x‘18’, x‘19’, x‘1B’, and x‘1D’ must be initialized to zero by privileged software before a new context is started in an SPE.

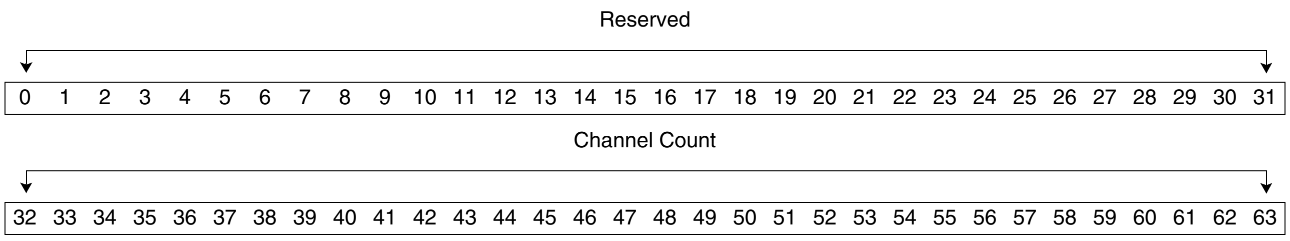
16.3.3 SPU Channel Count Register (SPU_ChnlCnt)

Each channel has a data port and an associated depth or count (that is, the number of entries in the channel). The channel count value is a record of the number of entries in the channel. The SPU Channel Count Register is used to read or initialize the count associated with the channel selected by the SPU Channel Index Register.

Access to this register should be privileged.

Channels are also defined as blocking or nonblocking. Blocking channels stall the execution of the SPU if the channel is full (on writes) or empty (on reads); that is, the SPU will stall with a channel count of zero.

Access Type	Read/Write
Base Address Offset	(BP_Base I P2(n)) + x‘04068’; where n is an SPE number.



Bits	Field Name	Description
0:31	Reserved	Set to zeros.
32:63	Channel Count	This field is used to modify the count parameter for the channel identified by the SPU Channel Index Register. The number of bits implemented for this field is channel specific (see the channel descriptions for the maximum value of the count). See the specific implementation documentation for more information.

Implementation Note:

The following channels have counts that must be initialized or restored through this interface:

- SPU Pending Event Register (supporting SPU Read Event Status Channel x‘0’)
- SPU Write Event Mask Channel (x‘1’)
- SPU Signal Notification 1 Channel (x‘3’)
- SPU Signal Notification 2 Channel (x‘4’)
- MFC Read Tag-Group Status Channel (x‘18’)
- MFC Read List Stall-and-Notify Tag Status Channel (x‘19’)
- MFC Read Atomic Command Status Channel (x‘1B’)
- SPU Read Inbound Mailbox Channel (x‘1D)

Note: Channel counts for channels x‘0’, x‘3’, x‘4’, x‘18’, x‘19’, x‘1B’, and x‘1D’ must be initialized to zero. Channel counts for channels x‘17’, x‘1C’, and x‘1E’ must be initialized to one. The channel count for the MFC Command Opcode Channel (see page 117) must be initialized to an implementation-dependent maximum value by privileged software before a new context is started.

16.4 SPU Configuration Register (SPU_Cfg)

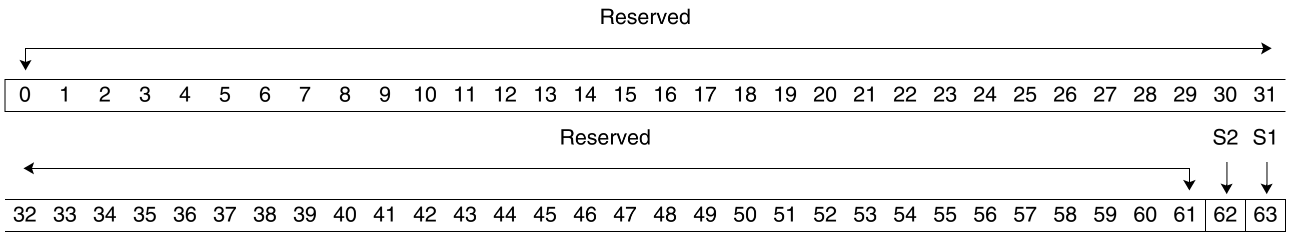
The SPU Configuration Register is used to read or set the configuration of the SPU signal notification registers: SPU Signal Notification 1 Register (see page 106) and SPU Signal Notification 2 Register (see page 107) in the SPUs.

Each SPU signal notification register can be configured to either overwrite the current contents of the register when written or to logically OR the data written with the current contents. The current contents are reset to zero when read using an SPU read channel (**rdch**) instruction.

The mode of each SPU signal notification register must be initialized to overwrite at power-on reset (POR).

Access Type Read/Write

Base Address Offset (BP_Base | P2(n)) + x‘04078’; where n is an SPE number.



Bits	Field Name	Description
0:61	Reserved	Reserved.
62	S2	Signal notification 2 mode. 0 Signal notification 2 mode is overwrite (POR default). 1 Signal notification 2 mode is logical OR.
63	S1	Signal notification 1 mode. 0 Signal notification 1 mode is overwrite (POR default). 1 Signal notification 1 mode is logical OR.

17. SPE Context Save and Restore

Saving and restoring the context of a Synergistic Processor Element (SPE) can be very expensive in terms of time and system resources. An SPE allocation scheme can be used that follows a “serially reusable device” model. In this model, an SPE is assigned a task until it completes, and then another task is assigned in a serial fashion. This model results in better utilization of SPEs. The serially reusable device model requires less context to be saved and restored during a context switch.

When this model cannot be used, the hardware supports suspending a task on an SPE, fully saving its context, fully restoring that context at a later time, and resuming the task. Although it is facilitated by hardware, context save and restore is software intensive. A preemptive context switch facilitated by privileged software 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 information must be performed. An application yielding mechanism can be used, in which the application using the SPU determines the timing and the amount of context saved and restored. This can provide significant cost savings in terms of the cycles and space that are used to save and restore the SPE context. However, the application primarily drives the technique selected; the architecture requires an implementation to support a full context save and restore of an SPE. Context Save and Context Restore is implementation dependent. See the specific implementation documentation for details on the required sequence for saving and restoring a context. Some operating environments can forgo restoring the context of an SPE that has been stopped (especially when stopped on error conditions).

Note: Preemptive context switching of an SPU that interfaces directly with an I/O device should be specifically avoided because the physically mapped local storage is considered part of the context.

The context save and restore sequences are described in the specific implementation documentation.

18. PPE Address Range Facility

The PowerPC Processor Element (PPE) address range facility provides a method to select a class ID for the cache replacement management facility based on the effective address of PPE loads and stores and instruction fetches. For more information about the cache replacement management facility, see *Section 19.1 Replacement Management Table Example* beginning on page 255. A set of PPE address range registers is provided in each PPE to implement this facility.

An implementation should provide a minimum of two sets of range registers for instruction fetches and two sets of range registers for data fetches per logical PPE. PPE special purpose registers (SPRs) are provided for these four range registers. The SPRs should be duplicated for each processor thread.

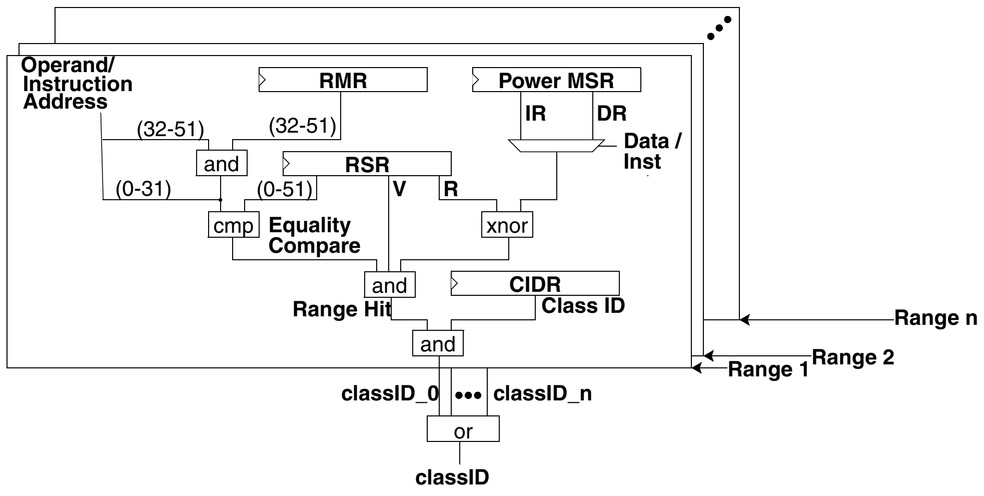
A PPE address range facility consists of the following registers:

- Range Start Register (see page 251)
- Range Mask Register (see page 252)
- Class ID Register (see page 253)

An address range is a naturally-aligned range that is a power of 2 in size and is between 4 KB and 4 GB, inclusive. An address range is defined by two registers; the Range Start Register (RSR) and the Range Mask Register (RMR). For each address range, there is an associated Class ID Register (CIDR) that specifies the class ID. These address range registers are accessible by PPE move-to or move-from special-purpose register instructions. Access to these registers is only allowed in privileged state.

Figure 18-1 illustrates how the address range registers are used to generate the class ID.

Figure 18-1. Generation of Class ID from the Address Range Registers



If all the following conditions are met, the particular address range defined by an RSR and RMR pair applies to a given effective address (EA), and a “range hit” is said to have occurred. Then the class ID from in the corresponding Class ID Register is used as an index into the RMT. For example:

- 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]
- Else (the operation is an instruction fetch)
  - RSR[62] = MSR[IR]

If there is no “range hit” for a given effective address, the class ID has a value of 0. In effect, the RMR defines the size of the range by selecting the bits of an effective address used to compare with the RSR.
The upper bits of an RSR contain the starting address of the range. The lower bits contain a relocation mode (virtual or real) and a valid bit. The size of the range must be a power of two. The starting address of the range must be a range-size boundary.

Note: All cache management instructions should be treated as loads or stores for calculating the class ID.

Programming Note:

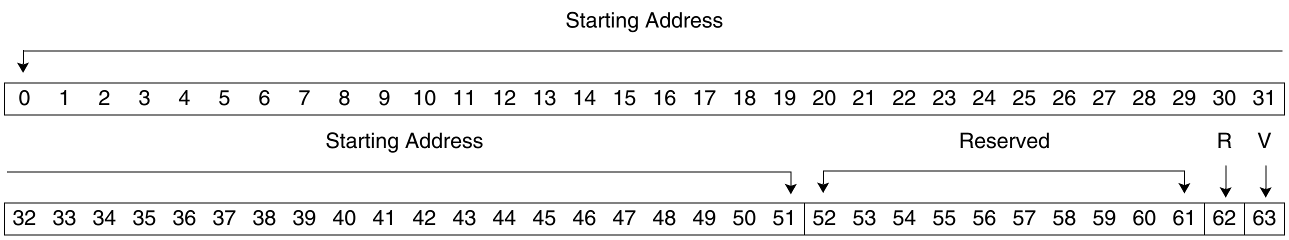
To avoid confusion about the class ID value, software should ensure that the address ranges specified in the PPE address range facility do not overlap (that is, more than one range has a simultaneous hit). If two address ranges are hit by the same address, the resulting class ID will be a logical OR of the two values in the CIDR.

18.1 Range Start Register (RSR)

The Range Start Register contains the starting address of a PPE address range. The upper bits of this register contain the starting address of the range. The lower bits contain a relocation mode (real or virtual) and a valid bit. The size of the range must be a power of two, and the starting address must be a range-size boundary.

Access Type Read/Write

SPR Offset Implementation dependent



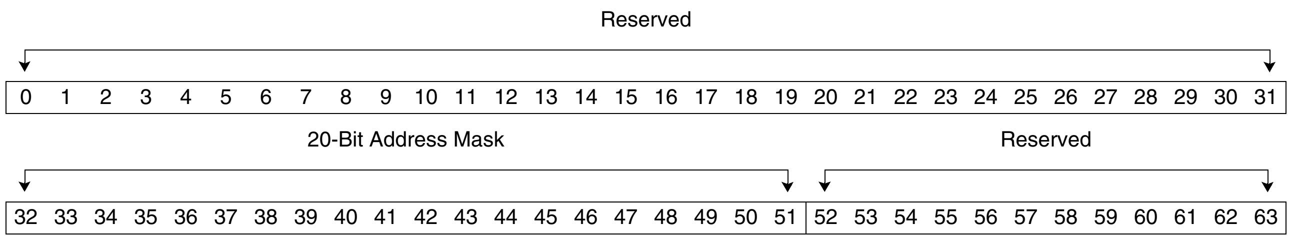
Bits	Field Name	Description
0:51	Starting Address	This field contains the upper 52 bits of the starting address for the range.
52:61	Reserved	Set to zero.
62	R	Relocation mode. Used to compare the Instruction Relocate (IR) or Data Relocate (DR) bit of the PPE Machine State Register (MSR). 0 Address relocation through effective-to-virtual-address translation is off (real addressing mode). 1 Address relocation through effective-to-virtual-address translation is on (virtual addressing mode).
63	V	Range register valid. 0 Range register disabled. 1 Range register enabled.

18.2 Range Mask Register (RMR)

The Range Mask Register defines the size of a PPE address range by selecting the bits of an operand address used to compare with the Range Start Register.

Bits 32 - 51 of the operand or instruction address is ANDed with the 20-bit RMR. Bits 0 - 31 of the operand or instruction address is then concatenated with the result of the AND and compared with the starting address in the RSR to determine a range hit. The upper 32 bits are always compared.

Access Type	Read/Write
SPR Offset	Implementation dependent



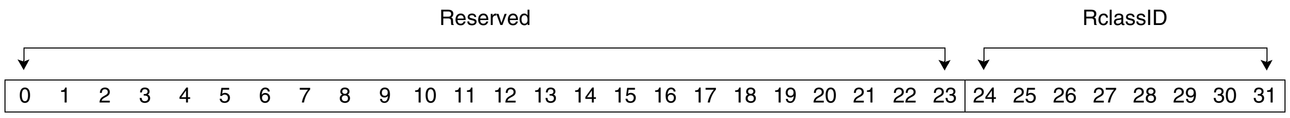
Bits	Field Name	Description
0:31	Reserved	Set to ‘1’. This reserved field is *not* set as described in *Section 1.5 Reserved Fields and Registers* beginning on page 32.
32:51	20-Bit Address Mask	Corresponds to effective address bits 32 - 51.
52:63	Reserved	Set to zeros.

18.3 Class ID Register (CIDR)

The Class ID Register contains the RclassID to use when the address of the operand or instruction matches a PPE address range.

Access Type Read/Write

SPR Offset Implementation dependent



Bits	Field Name	Description
0:23	Reserved	Reserved.
24:31	RclassID	The size of this field is implementation dependent. See the specific implementation documentation for more information.

19. Cache Replacement Management Facility

The Cell Broadband Engine Architecture (CBEA) provides a way to control cache replacement based on a replacement class identifier (RClassID). The class ID is a parameter in the memory flow controller (MFC) commands and is generated from the load and store address for PowerPC Processor Element (PPE) operations (see *Section 18* on page 249). The class ID is used to generate an index to a table managed by privileged software that is used to control the replacement policy.

A CBEA-compliant implementation should provide an RMT for each major cache structure. The format of the replacement management table (RMT) is implementation dependent. An example of an RMT and the index generation method is provided in the following sections. See the specific implementation documentation for details on the support of the cache replacement management facility.

In this version of the CBEA, memory-mapped I/O (MMIO) register locations are provided for an L2 RMT and an MFC translation lookaside buffer (TLB) RMT. PPE special purpose registers are provided for a PPE TLB RMT.

19.1 Replacement Management Table Example

Privileged software controls cache replacement through an RMT. Each level of cache, including the TLBs that support replacement management, must have an independent RMT.

The RMT consists of an implementation-dependent number of entries, which should contain Set-Enable bits, a Valid bit, and other control information. Optionally, an implementation can also provide a Cache Bypass bit and an Algorithm bit. The number of entries and the size of each entry in the RMT table is implementation dependent. *Table 19-1* depicts a typical RMT entry for an 8-way, set-associative cache. The RMT table is located in the real address space of the system. The privileged software should map these RMT tables as privileged pages.

Table 19-1. Typical RMT Entry for an 8-Way Set Associative Cache

S0	S1	S2	S3	S4	S5	S6	S7	Reserved																					a	b	v
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
0:7	S(n)	Cache set enable: n = {0 - 7}.
8:28	Reserved	
29	a	The Algorithm bit specifies the replacement algorithm to be used for this class. 0 Least recently used (LRU) 1 Most recently used (MRU)
30	b	The Bypass bit indicates that the operation should not be cached at this level (not valid for translation RMTs).
31	v	The Valid bit indicates that the RMT entry contains valid information.

The RMT defines which sets in the set-associative cache are to be used for the respective replacement management class. If the Set Enable bit for the respective set is ‘1’, that set is used by that replacement class in the RMT entry. If the Set Enable bit is not set, the associated set is not used for operations involving the respective replacement management class.

One or more sets can be used for more than one replacement management class. Using replacement management classes for streaming areas to prevent thrashing[1] the cache is an example of how the cache replacement management facility can be used.

Accessing an invalid RMT results in the default class of x'0' being used for the operation. If the default class is also invalid, the class is treated as though all set enables were set.

Setting the Bypass bit indicates that the data should not be cached at a hierarchy level that corresponds to this RMT. For data caches, the load or store should bypass the cache and be passed directly to the bus. It is possible that the data for the operation already exists at this hierarchy level. In this case, the implementation can source the data from the cache, provide the data through intervention, or cause the data to be invalidated or pushed from this cache level. In any case, the load or store still follows the normal rules for coherency. A table is generated to show all possible conditions for WIMG[2] settings and cache states.

19.2 RMT Index Generation Example

The RMT is indexed for two purposes: to update the contents and to access the contents for management of direct memory accesses (DMAs) and loads and stores.

To update the contents, software sets the RMT Index Register to point to the entry and stores the new data for the RMT entry to the RMT Data Register. Accessing the RMT for management purposes is system dependent. The CBEA places few requirements on the index. Devices that share a cache hierarchy must be able to share the entries in the RMT (if the RMT index is the same for two or more devices), or must have an independent area of the table (if the RMT index is unique).

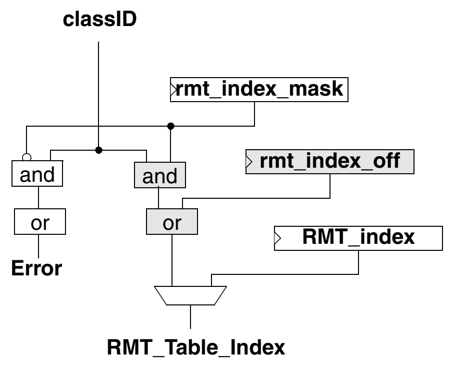
When the cache hierarchy level is dedicated to a single device, the RMT index can be as simple as a range check on the class ID. In the case of a shared cache, the class ID must be converted to an RMT index. *Figure 19-1 RMT Index Generation* on page 257 illustrates one method of generating the RMT from the class ID. In this example, a pair of registers is used to map the class ID to an RMT index. The RMT Index Mask Register is used to mask off the upper bits of the class ID. The RMT Index Off Register replaces the bits disabled by the RMT Index Mask Register. Each device that shares this cache hierarchy level must have an independent set of mask and offset registers. The shaded items are only required if more than one device shares the RMT.

Access to the RMT Index Mask and RMT Index Off Registers should be privileged. Privileged software must set the RMT Index Mask Register with zeros in the upper bits and ones in the lower bits. The number of zeros in the lower bits of the RMT Index Off Register must be equal to or greater than the number of ones in the RMT Index Mask Register.

The RMT Index Register and the RMT Data Register are implementation dependent. A general format for these registers is provided in *Section 19.2.1 RMT Index Register* on page 257 and *Section 19.2.2 RMT Data Register* on page 258. See the specific implementation documentation for more information.

1. A cache is said to thrash when its miss rate is too high and it spends most of its time servicing misses.
2. The 4 bits in the page table, also called a page table entry, that control processor accesses to cache and to main storage. "W" stands for write through, "I" for caching inhibited, "M" for memory coherence, and "G" for guarded storage.

Figure 19-1. RMT Index Generation



19.2.1 RMT Index Register (RMT_Index)

This register holds the index of the RMT entry that is to be modified using the RMT Data Registers. The RMT Index Register is an optional facility. An RMT Index Register must be provided for each facility that supports the cache replacement policy.

Table A-3 SPE Privilege 1 Memory Map on page 298 lists the address offsets for the Synergistic Processor Element (SPE) TLB RMTs. *Table A-5 PPE Privilege 1 Memory Map* on page 302 lists the address offsets for the PPE L2 RMTs. *Table C-1 PPE Special Purpose Register Map* on page 309 lists the special-purpose register (SPR) numbers for the PPE TLB RMTs. See the specific implementation documentation for more information about the implementation of the RMT facility.

Access Type	Read/Write
SPR Offset	Implementation dependent
Base Address Offset	Implementation dependent

Implementation Dependent

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

Index

32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63

Bits	Field Name	Description
0:31	Implementation Dependent	
32:63	Index	The number of bits in this field is implementation dependent.

19.2.2 RMT Data Register (RMT_Data)

This register is used to access the entry in the RMT pointed to by the RMT Index Register. The RMT Data Register is an optional facility. An RMT Data Register must be provided for each facility that supports the cache replacement policy.

Table A-3 SPE Privilege 1 Memory Map on page 298 lists the address offsets for the SPE TLB RMTs. *Table A-5 PPE Privilege 1 Memory Map* on page 302 lists the address offsets for the PPE L2 RMTs. *Table C-1 PPE Special Purpose Register Map* on page 309 lists the address offsets for the PPE TLB RMTs. See the specific implementation documentation for more information about the implementation of the RMT facility.

Access Type	Read/Write
SPR Offset	Implementation dependent
Base Address Offset	Implementation dependent

Implementation Dependent																															
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

Implementation Dependent																															
32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63

Bits	Field Name	Description
0:63	Implementation Dependent	Implementation dependent

20. Resource Allocation Management

Resource allocation management (RAM) provides a mechanism to allocate portions of a resource's time to a specific resource allocation group (RAG). The number of RAGs is implementation dependent. The resources that are managed are memory units and I/O interfaces (IOIFs). The definition of a memory unit is implementation dependent, and could, for example, be defined as a range of addresses or interleaved addresses. The resource allocation groups are groups of one or more hardware units called requesters. These requesters are physical or virtual units; they can initiate load or store requests, or they can initiate DMA read or write accesses. The requesters are:

- PowerPC Processor Element (PPE) groups (PPE 0, PPE 1, memory management unit [MMU], L2 cache)
- Synergistic Processor Elements (SPEs)
- Virtual channels associated with the physical IOIFs

PPE groups and the SPEs are each in a specific RAG at an instant in time. However, they can be assigned to various RAGs over time. Each RAG can be identified by a resource allocation ID (RAID). Software can configure the RAID for a PPE group or an SPE with a memory-mapped register.

See the specific implementation documentation for more information.

21. Interrupt Facilities

The Cell Broadband Engine Architecture (CBEA) provides facilities for the following activities:

- Routing interrupts and interrupt status information to a PowerPC Processor Element (PPE) or external devices
- Prioritizing interrupts presented to a PPE
- Generating an interprocessor interrupt

In the CBEA, interrupts are a result of application actions, errors, or unusual conditions that arise in the execution of PPE or synergistic processor unit (SPU) instructions, the execution of memory flow controller (MFC) commands, or the occurrence of external events. Interrupts directed to a PPE allow the PPE to change state as a result of the interrupt.

The CBEA defines additional interrupting conditions for the Synergistic Processor Elements (SPEs). Each SPE has a set of interrupt registers for masking the interrupting conditions, holding the status of the interrupting conditions, and routing the interrupt to a PPE or other device in the system.

PowerPC Architecture, Book III describes the interrupt definitions, interrupt ordering, interrupt synchronization, and interrupt processing provided under the PowerPC Architecture. The PowerPC Architecture defines a condition that can cause an interrupt as an exception. An interrupt is the change in processor state that is caused by handling an exception.

When enabled, an SPE interrupt condition causes an interrupt to be routed to a PPE or other device which, depending on the state of the PPE, can cause a change in the processor's state. Interrupts routed to a PPE are presented as external interrupts.

Note: By masking the interrupting condition, privileged software can also support polling.

21.1 Interrupt Classification

In the PowerPC Architecture, interrupts are classified by cause. An interrupt that is directly caused by the execution of an instruction is an “instruction caused” interrupt. All other system exceptions cause “system caused” interrupts. The additional interrupting conditions defined by the CBEA are classified as system caused interrupts. See *Section 21.4 SPU and MFC External Interrupt Definitions* on page 269 for the CBEA-defined interrupt conditions and the CBEA-defined interrupt classes. The PowerPC interrupt classifications should not be confused with the CBEA-defined interrupt classes.

System-caused interrupts are presented to a PPE as external interrupts. External interrupts are always imprecise with respect to instruction processing in a PPE, and they cause an asynchronous change in state. See *PowerPC Architecture, Book III* for a description of external interrupts.

External interrupts are maskable in a PPE, and a mask for each additional interrupt condition is supported in the CBEA. See *Section 21.6 MFC Interrupt Mask Registers* beginning on page 276 for a description of external interrupt masks. An implementation can provide additional implementation-dependent interrupt status and mask registers. See the specific implementation documentation for more information.

21.2 Interrupt Presentation

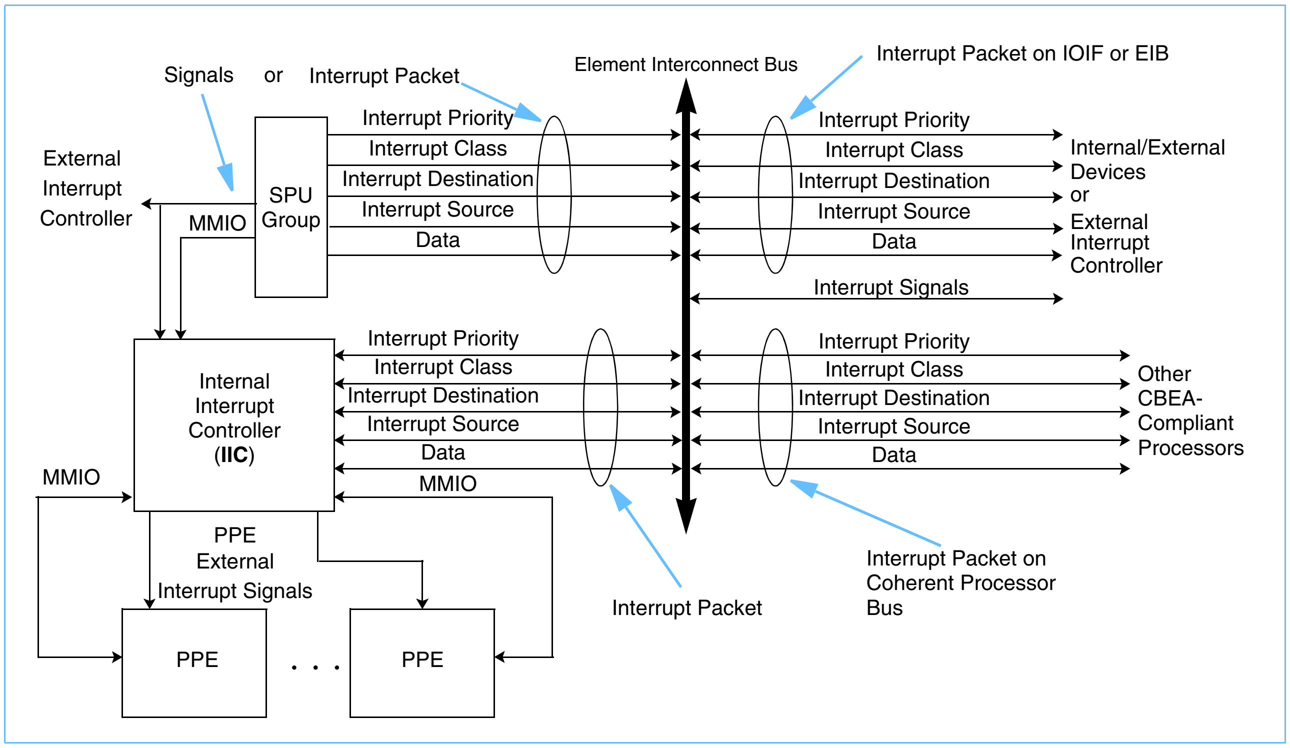
The PPEs in a CBEA-compliant processor can generate and service interrupts and can handle interrupts generated by SPUs, MFCs, and other external devices. Interrupts generated by an SPU instruction are routed to a PPE or to other devices for processing. Software can generate an interrupt to a PPE. SPUs can also generate and service interrupts that result from SPE events.

Figure 21-1 illustrates how the interrupts are presented inside a CBEA-compliant processor.

Interrupts generated by an SPU group are sent to an external interrupt controller or to an internal interrupt controller (IIC) using either dedicated signals or an interrupt packet on the internal element interconnect bus (EIB). External devices send interrupts to a CBEA-compliant processor using either dedicated signals or an interrupt packet to the I/O interface (IOIF), which forwards the interrupt to the IIC through the EIB. An IIC receives the interrupt packet and signals an external interrupt to the appropriate PPE. Software running on a PPE or an SPU can cause an interrupt to be sent to a logical PPE by writing the corresponding Interrupt Generation Port Register. As a result of the MMIO write, the IIC signals an interrupt to a PPE.

The IIC interrupt generation, routing, and presentation are described in *Section 21.3 Internal Interrupt Controller Registers* beginning on page 263. SPU group registers related to external interrupt generation, routing, and presentation are described in *Section 21.4 SPU and MFC External Interrupt Definitions* beginning on page 269 through *Section 21.8 Interrupt Routing Register* beginning on page 283.

Figure 21-1. Interrupt Presentation


Signals
or
Interrupt Packet
Element Interconnect Bus
Interrupt Packet on IOIF or EIB
External Interrupt Controller
MMIO
SPU Group
Interrupt Priority
Interrupt Class
Interrupt Destination
Interrupt Source
Data
Internal/External Devices or External Interrupt Controller
Interrupt Signals
Internal Interrupt Controller (IIC)
MMIO
Other CBEA-Compliant Processors
PPE External Interrupt Signals
PPE
. . .
PPE
Interrupt Packet
Interrupt Packet on Coherent Processor Bus

21.3 Internal Interrupt Controller Registers

The IIC has an interrupt control block for each logical PPE (that is, for each thread in the physical processor). These control blocks are mapped in the real address space, starting at an implementation-dependent offset from the BP_Base. The control block's starting address is defined as the BP_Base | IIC (p) + x'400' + (t × x'20'); where 'p' is the physical PPE, and 't' is the PPU thread number for the corresponding PPE. (See *Table A-1 CBEA-Compliant Processor Memory Map* on page 294 for more details.)

Table 21-1 shows the registers associated with each control block and their offsets from starting address of the BP_Base.

Table 21-1. Internal Interrupt Controller Memory Map

Offset (Hexadecimal)	Register	Description	Access Type
PowerPC Processor Unit (PPU) (Thread t) Interrupt Control Block, where 0 ≤ t ≤ number of PPU threads -1			
x'000'	INT_Pending_NonD	Interrupt Pending Port Registers; nondestructive read. Status and data for pending interrupt.	Read Only
x'008'	INT_Pending_D	Interrupt Pending Port Registers; destructive read. Status and data for pending interrupt.	Read Only
x'010'	INT_Generation	Interrupt Generation Port Register (see page 267). Port for generation of an interprocessor interrupt.	Write Only
x'018'	INT_CPL	Interrupt Current Priority Level Register (see page 268). Only higher-priority interrupts cause an external interrupt.	Read/Write

21.3.1 Interrupt Pending Port Registers (INT_Pending_NonD and INT_Pending_D)

The Interrupt Pending Port Registers allow software to read the interrupt packet data and other information about the highest priority interrupt pending for each logical PPE. There is one Interrupt Pending Port Register for each logical PPE in a CBEA-compliant processor. The following description applies to interrupts for one logical PPE and the Interrupt Pending Port Register, Interrupt Current Priority Level Register, and Interrupt Generation Port Register for that logical PPE.

When an interrupt packet arrives at the IIC, if an interrupt of the same priority is not already queued in the IIC for the PPE, the interrupt packet is retained in an interrupt pending queue. It is implementation dependent whether more than one interrupt packet of the same priority is retained or whether subsequent interrupt packets are retried. However, the IIC implementation must always be able to accept at least one interrupt per priority value.

When the priority of the highest priority, valid interrupt in the interrupt pending queue is higher (lower numeric value) than the priority in the Interrupt Current Priority Level Register, the external interrupt signal to a PPE 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 Interrupt Current Priority Level Register, the external interrupt signal to the PPE is deactivated.

Software should read the Interrupt Pending Port Register in the first level interrupt handler (FLIH) through an MMIO load to obtain information about the interrupting condition. When software reads the Interrupt Pending Port Register, the IIC returns the value of the highest priority, valid interrupt in the interrupt pending queue. Software can read the Interrupt Pending Port Register with either of two addresses. One address is used for destructive reads, and the other address is used for nondestructive reads. When the Interrupt Pending Port Register is read destructively, the Interrupt Valid bit for the highest priority, valid interrupt in the interrupt pending queue is set to ‘0’. The priority value of this interrupt is copied into the Interrupt Current Priority Level Register. Thus, a destructive read causes the external interrupt signal to a PPE to be deactivated because the Interrupt Current Priority Level Register value represents the highest priority of the pending interrupts. This external interrupt signal to a PPE remains deactivated until one of two events occurs. A higher priority interrupt packet arrives at the IIC, or software lowers the priority in the Interrupt Current Priority Level Register and a higher priority interrupt packet is or becomes pending. When the Interrupt Pending Port Register is read nondestructively, the Valid bit in the interrupt pending queue is not changed, and the Interrupt Current Priority Level Register is not changed.

When the Interrupt Pending Port Register is read (destructively or nondestructively) and there is a valid interrupt in the interrupt pending queue, the data returned for the highest priority interrupt includes the following information about the interrupt: validity, type, class, interrupt source (ISRC), priority and, optionally, the interrupt packet data. If an implementation does not support the interrupt packet data, the interrupt packet data is returned as zeros. If the Type bit is ‘0’, the interrupt packet originated from a memory flow controller (MFC), external device, or external interrupt controller. If the Type bit is ‘1’, the interrupt packet originated from the Interrupt Generation Port Register, and the class information and ISRC are read as zeros.

If software reads the interrupt pending port (destructively or nondestructively) when there is no valid pending interrupt, the Valid bit returned will be zero. The value returned for the other fields is undefined. If software reads the interrupt pending port when there is no valid pending interrupt, the Interrupt Current Priority Level Register is unchanged.

If a nondestructive read of the interrupt pending port that returns a valid interrupt is followed by another read of the interrupt pending port (destructive or nondestructive), the IIC returns the same data value unless another interrupt of a higher priority has been received by the IIC. For example, there are two pending interrupts of the same priority: one due to an external device sending an interrupt packet, and one due to a store to the interrupt generation port (IGP). Once software reads the interrupt pending port nondestructively and gets the interrupt pending port value for one of these interrupts, the same interrupt pending port value is returned on the next interrupt pending port read, assuming no interrupt of a higher priority has been received by the IIC.

When a valid interrupt of a specific priority exists in the interrupt pending queue, other pending interrupts of the same priority and destination can be queued at other points in a CBEA-compliant processor or externally. Reading the Interrupt Pending Port Register destructively resets the Interrupt Valid bit, allowing a subsequent interrupt to move into the interrupt pending queue. The latency for a pending interrupt to move to the Interrupt Pending Port Register is implementation dependent.

An implementation can support fewer bits than architected for interrupt packet data, interrupt class, ISRC, and priority. If an implementation supports fewer priority bits than the number architected, the most-significant bits of the Priority field should be supported to get a more consistent view of priority in an environment where implementations or external devices support different numbers of bits. An implementation with N priority bits divides the full set of 256 priority values into 2^N ranges, so that priorities from implementations with all priority bits that fall in the different 2^N ranges still appear in their priority order with respect to each other.

If software synchronizes the MMIO load for a destructive read of the interrupt pending port, and then sets the Machine State Register (MSR) External Interrupt Enable bit to ‘1’, no external interrupt will occur at the same or a lower priority until software directly or indirectly modifies the Interrupt Current Priority Level Register.

Implementation Note:

For the deactivation of the external interrupt property to function correctly, hardware must ensure that the data read from the interrupt pending port is always returned before the external interrupt signal is deactivated.

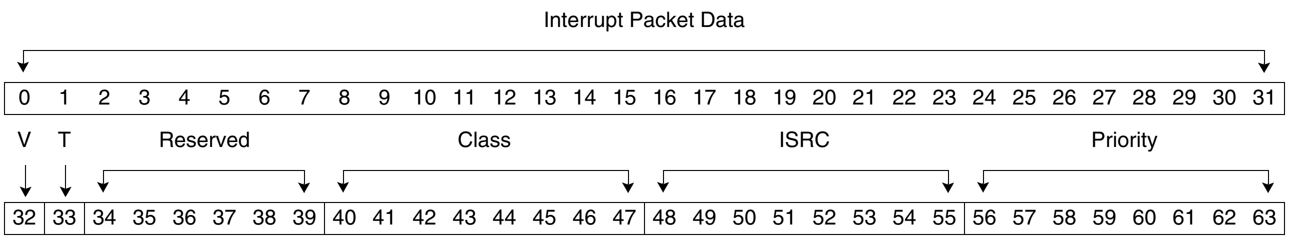
The interrupt pending port acts as a window into the interrupt pending queue. Software always sees the highest priority interrupt in this window at the time the interrupt pending port is read. This view of the interrupt pending queue is independent of the Interrupt Current Priority Level Register (see page 268).

After a valid interrupt is pending in the interrupt pending queue, software can cause the external interrupt signal to a PPE to be activated or deactivated by writing the Interrupt Current Priority Level Register. Software can also mask interrupts from the IIC by writing the current priority level value to 0.

A flat interrupt priority scheme, in which all interrupts have the same priority, can be implemented as follows: Set the same priority value for all classes in the Interrupt Routing Register. Configure the same priority value for interrupts associated with external devices and any external interrupt controller. After the destructive read of the Interrupt Pending Port Register, which software performs for each occurrence of a PPE external interrupt, software must store a lower priority (higher numeric value) to the Interrupt Current Priority Level Register to enable the presentation of the next interrupt packet.

Access Type Read Only

Base Address Offset BP_Base | IIC(p) + x‘400’ + (t × x‘20’) + x‘000’; Nondestructive read
BP_Base | IIC(p) + x‘400’ + (t × x‘20’) + x‘008’; Destructive read
(where p is a PPE number and t is a PPE (p) thread number)



Bits	Field Name	Description
0:31	Interrupt Packet Data	The meaning of the data in this field is dependent on the interrupt source. (This field is optional.)
32	V	Interrupt valid.
33	T	Interrupt type. 0 Interrupt from MFC, external device, or external interrupt controller. 1 Interrupt from interrupt generation port.
34:39	Reserved	Reserved.

Bits	Field Name	Description
40:47	Class	Interrupt class. If the SPE is the source of the interrupt, the following values are defined for the field: 0 Error 1 Translation 2 Application 3 Reserved If a PPE or the external interrupt controller is the source of the interrupt, the values defined for this field are implementation dependent. The first level interrupt handler uses the Class and ISRC fields to route the interrupt to the appropriate second level interrupt handler.
48:55	ISRC	Interrupt source. Defines the unit in the processor sourcing the interrupt. The values defined for this field are implementation dependent. The first level interrupt handler uses the Class and ISRC fields to route the interrupt to the appropriate second level interrupt handler.
56:63	Priority	Interrupt priority.

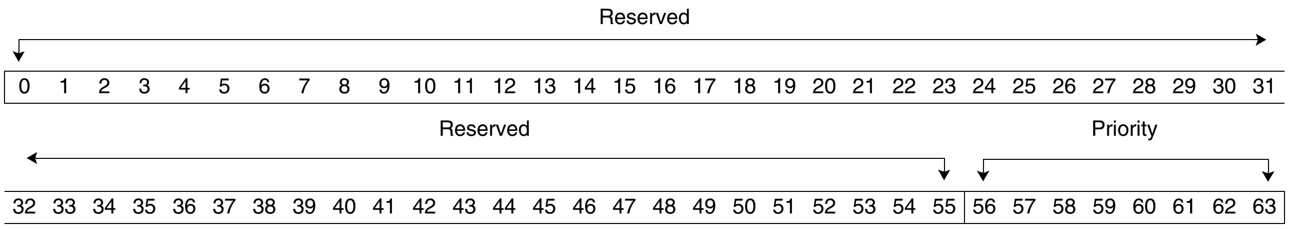
21.3.2 Interrupt Generation Port Register (INT_Generation)

The Interrupt Generation Port Register allows privileged software to generate an interrupt packet to a PPE. There is one Interrupt Generation Port Register for each PPE in a CBEA-compliant processor. Software can generate an interrupt packet to a PPE by storing to the PPE Interrupt Generation Port Register. This interrupt packet does not need to be transmitted on the internal CBEA-compliant processor interconnect bus, because the Interrupt Generation Port Register and the destination of the interrupt packet are both in the same IIC. The least-significant byte written to this register contains the interrupt priority. When the interrupt packet is read through the Interrupt Pending Port Register, the interrupt packet data, class information, and ISRC are read as zeros.

If there are multiple stores to a PPE interrupt generation port with the same priority value, and thus multiple pending interprocessor interrupts to a PPE with the same priority, interrupt packets of the same priority can be merged and only a subset of these interrupts presented.

An implementation can support fewer bits than the architecture provides for priority. If an implementation supports fewer priority bits than the number provided, the supported bits must be in the most-significant bit positions of the Priority field.

Access Type	Write Only
Base Address Offset	BP_Base | IIC(p) + x‘400’ + (t × x‘20’) + x‘010’ (where p is a PPE number and t is a PPE (p) thread number)



Bits	Field Name	Description
0:55	Reserved	Reserved.
56:63	Priority	Interrupt priority.

Programming Note:

For interprocessor interrupts, software might need to use other means, such as a message queue in memory, to convey information such as class, interrupt source, and the reason for the interrupt. Because a subset of interrupts of the same priority to a PPE might be presented through the interrupt pending port, a message queue in memory can also convey information about individual interrupts with the same priority.

21.3.3 Interrupt Current Priority Level Register (INT_CPL)

There is one Interrupt Current Priority Level Register for each PPE in a CBEA-compliant processor. It holds the priority level at which software is currently operating. The lower the numeric value of the Priority field, the higher the priority level. Thus, the highest priority corresponds to a numeric value of 0. The priority level value can be written explicitly by software storing to the Interrupt Current Priority Level Register, or indirectly by software performing a destructive read of the interrupt pending port. See *Section 21.3.1 Interrupt Pending Port Registers* on page 263 for details of interrupt pending port reads.

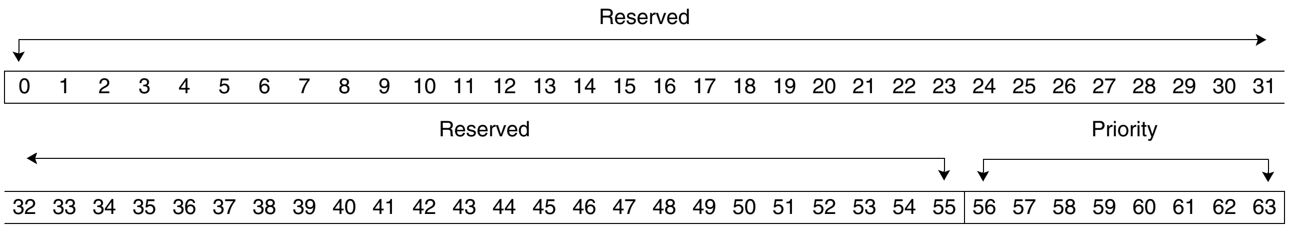
Software can mask interrupts at and below a specific priority level by storing the required priority value in the Interrupt Current Priority Level Register. To synchronize the interrupt masking side effects of this store, software must read the current priority level and synchronize the MMIO load for this read. After this synchronization, an external interrupt will not occur unless its priority is higher than the priority value in the Interrupt Current Priority Level Register.

Implementation Note:

For the deactivation of the external interrupt property to function correctly, hardware must ensure that the data read from the Interrupt Current Priority Level Register is always returned before the external interrupt signal is deactivated.

Access Type Read/Write

Base Address Offset BP_Base | IIC(p) + x‘400’ + (t × x‘20’) + x‘018’
(where p is a PPE number and t is a PPE (p) thread number)



Bits	Field Name	Description
0:55	Reserved	Reserved.
56:63	Priority	Current priority level (0 is highest priority).

21.4 SPU and MFC External Interrupt Definitions

Table 21-2 and *Table 21-3* show the additional interrupt definitions provided by the CBEA. These interrupt definitions share the same external interrupt vector defined in *PowerPC Architecture, Book III*. The interrupt definitions listed in *Table 21-2* and *Table 21-3* are generated by the MFC and SPU within a CBEA-compliant processor.

Table 21-2. SPU and MFC Interrupt Class Definitions

Class	Meaning	Description
0	Error	This interrupt class is used for all error conditions relating to SPU group processing and direct memory access (DMA) transfers.
1	Translation	This interrupt class is used for all translation exceptions relating to a DMA transfer.
2	Application	This interrupt class is used for all application interrupts.
3	Reserved	This interrupt class is reserved for future use.

Table 21-3. SPU and MFC External Interrupt Definitions (Page 1 of 2)

Class	Interrupt Type	Description
0	DMA Alignment Error Interrupt	This interrupt occurs when software attempts to execute a DMA command that is not aligned.
0	Invalid DMA Command Interrupt	This interrupt occurs when software attempts to execute a DMA 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.
0	SPU Error Interrupt	This interrupt occurs when the SPU has encountered one of the error conditions listed below: • Illegal channel instruction detected. • Invalid instruction detected. • Other implementation-dependent errors. (See the documentation for a specific implementation.) All these conditions should be individually maskable in the SPU.
1	MFC Data Segment Error Interrupt	This interrupt occurs when the DMA effective address cannot be translated to a virtual address.
1	MFC Data Storage Error Interrupt	This interrupt occurs when the DMA effective address cannot be translated to a real address.
1	MFC Local Storage Address Compare Suspend on **get** Interrupt	This interrupt occurs when MFC command queue operation is suspended (for both queues) due to an MFC Local Storage Address Compare match. A match occurs when the local storage address of an MFC DMA command matches the range specified in the MFC Local Storage Address Compare Register when writing to local storage and the MFC_ACCR[Lg] bit is set to '1'. See *Section 15.6* on page 227 and *Section 15.7* on page 229.
1	MFC Local Storage Address Compare Suspend on **put** Interrupt	This interrupt occurs when MFC command queue operation is suspended (for both queues) due to an MFC Local Storage Address Compare match. A match occurs when the local storage address of an MFC DMA command matches the range specified in the MFC Local Storage Address Compare Register when reading from local storage and the MFC_ACCR[Lp] bit is set to '1'. See *Section 15.6* on page 227 and *Section 15.7* on page 229
2	Mailbox Interrupt	This interrupt occurs when an SPU writes to an SPU Write Outbound Interrupt Mailbox Channel (see page 140).
2	SPU Stop-and-Signal Instruction Trap	This interrupt occurs when an SPU executes a stop-and-signal (**stop**) instruction.

Table 21-3. SPU and MFC External Interrupt Definitions (Page 2 of 2)

Class	Interrupt Type	Description
2	SPU Halt Instruction Trap or Single Instruction Step Complete	This interrupt occurs when an SPU executes a halt conditional instruction, and the condition is met. It also occurs after completing an instruction in single step mode when the SPU_PrivCntl[S] bit is set to '1' (see *Section 16.1 SPU Privileged Control Register* on page 239).
2	Tag-Group Completion Interrupt	This interrupt occurs when all commands for a selected tag group or tag groups are complete. The interrupt generation is dependent on the Proxy Tag-Group Query Mask Register (see page 94) and the Proxy Tag-Group Query Type Register (see page 93).
2	SPU Inbound Mailbox Threshold Interrupt	This interrupt occurs when the number of valid entries in the SPU inbound mailbox queue is below an implementation-dependent value or threshold. Valid entries are removed from the queue by an SPU application reading from the SPU Inbound Mailbox Channel. See *Section 8.6* beginning on page 101 for more information about the mailbox facility.

21.5 SPU and MFC Interrupt Generation Process

The following three classes of interrupts are generated:

- Class 0 Interrupts
- Class 1 Interrupts (see page 272)
- Class 2 Interrupts (see page 274)

21.5.1 Class 0 Interrupts

As shown in *Figure 21-2* on page 271, there are two SPU exceptions that cause the SE bit in the MFC Class 0 Interrupt Status Register to be set to ‘1’:

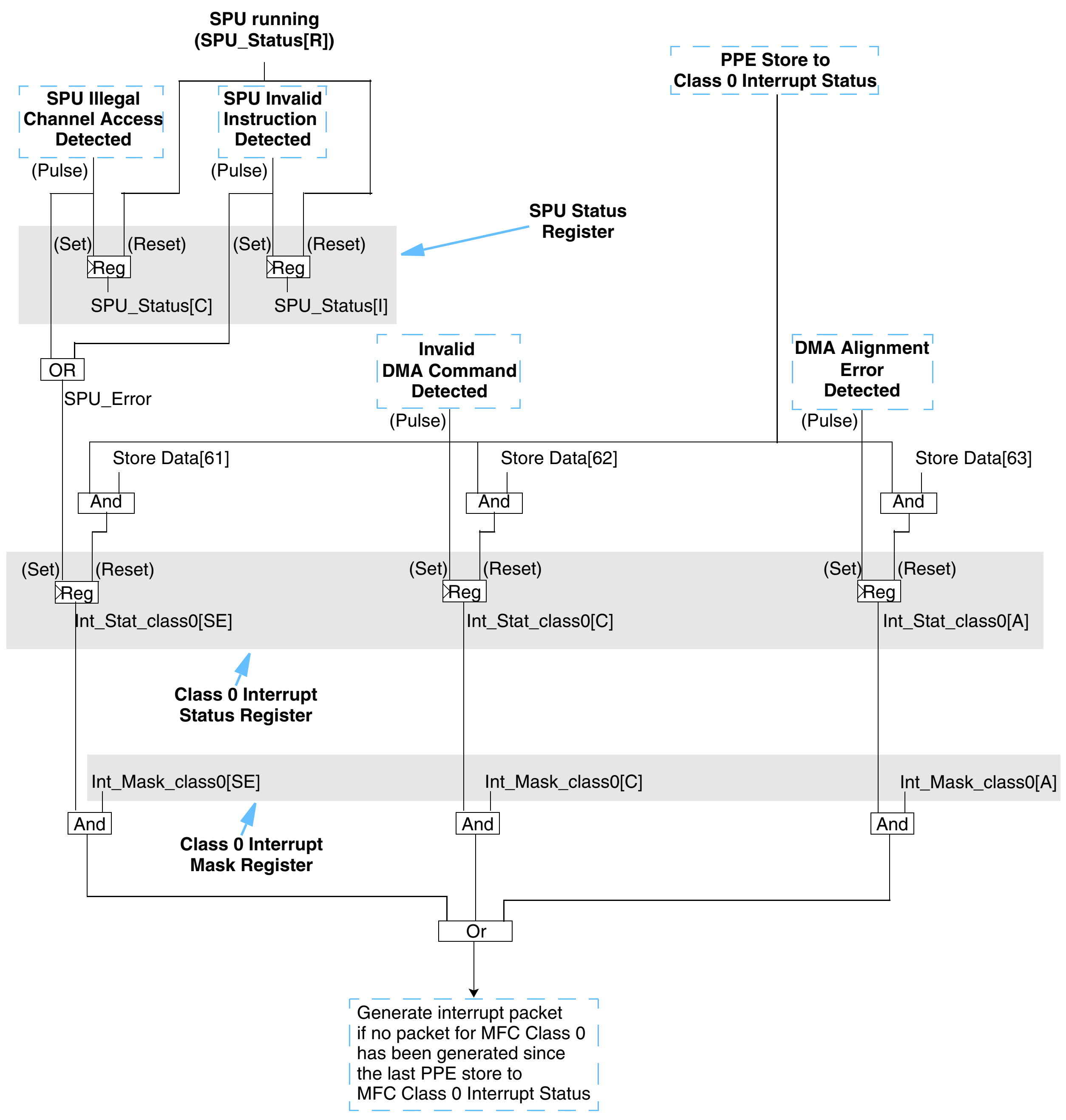
- If the SPU attempts an illegal channel access, the C bit of the SPU Status Register is set to ‘1’ and the SE bit is set to ‘1’.
- If the SPU attempts an invalid instruction, the I bit in SPU_Status is set to ‘1’ and the SE bit is set to ‘1’.

As shown in *Figure 21-2* on page 271, there are also two MFC exceptions that cause MFC class 0 Interrupt Status bits to be set:

- An invalid DMA command sets the C bit of the MFC Class 0 Interrupt Status Register to ‘1’.
- A DMA alignment error sets the A bit of the MFC Class 0 Interrupt Status Register to ‘1’.

Enabled bits in the MFC Class 0 Interrupt Status Register that are set to ‘1’ can cause a class 0 interrupt packet to be sent as described in *Section 21.8 Interrupt Routing Register* beginning on page 283.

Figure 21-2. MFC Class 0 Interrupt Generation Process


SPU running (SPU_Status[R])
SPU Illegal Channel Access Detected
SPU Invalid Instruction Detected
PPE Store to Class 0 Interrupt Status
(Pulse)
(Pulse)
SPU Status Register
(Set)
(Reset)
Reg
(Set)
(Reset)
Reg
SPU_Status[C]
SPU_Status[I]
OR
SPU_Error
Invalid DMA Command Detected
DMA Alignment Error Detected
(Pulse)
(Pulse)
Store Data[61]
Store Data[62]
Store Data[63]
And
And
And
(Set)
(Reset)
Reg
(Set)
(Reset)
Reg
(Set)
(Reset)
Reg
Int_Stat_class0[SE]
Int_Stat_class0[C]
Int_Stat_class0[A]
Class 0 Interrupt Status Register
Int_Mask_class0[SE]
Int_Mask_class0[C]
Int_Mask_class0[A]
And
And
And
Class 0 Interrupt Mask Register
Or
Generate interrupt packet if no packet for MFC Class 0 has been generated since the last PPE store to MFC Class 0 Interrupt Status

Note: It is undefined whether a “set” is set dominant or not, unless this is explicitly stated.

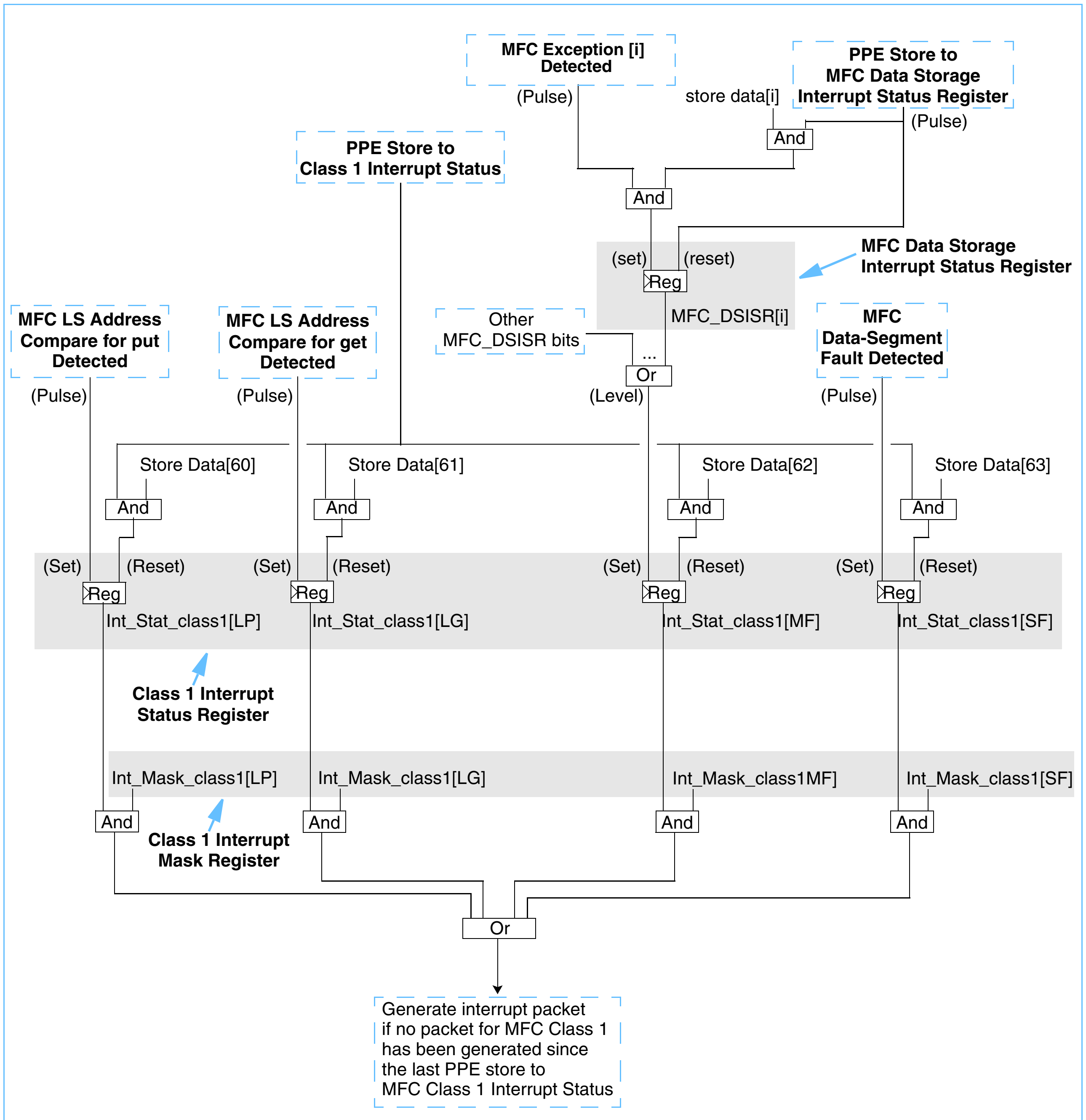
21.5.2 Class 1 Interrupts

Only MFC events can cause class 1 interrupts to be generated, as shown in *Figure 21-3* on page 273. There are several defined MFC exceptions that set the bits of the MFC Data Storage Interrupt Status Register to '1':

- If a bit in the MFC_DSISR is set to '1,' the MF bit of the MFC Class 1 Interrupt Status Register (see page 281) is set to '1'. In this case, the software interrupt handler needs to clear the bits in the MFC_DSISR by storing a '0' in the respective bit position before attempting to reset bit 62 of the MFC Class 1 Interrupt Status Register.
- If the MFC_DSISR bits are not cleared and the MF bit of the MFC Class 1 Interrupt Mask Register (see page 277) is set to '1', another interrupt packet is generated.
- If an MFC data-segment fault occurs, it sets the SF bit of the MFC Class 1 Interrupt Status Register to '1'.

Any enabled interrupts in the MFC Class 1 Interrupt Mask Register whose corresponding bit is set to '1' in the MFC Class 1 Interrupt Status Register can cause an interrupt packet to be sent as described in *Section 21.8 Interrupt Routing Register* beginning on page 283.

Figure 21-3. MFC Class 1 Interrupt Generation Process


MFC Exception [i] Detected
(Pulse)
store data[i]
PPE Store to MFC Data Storage Interrupt Status Register
(Pulse)
And
PPE Store to Class 1 Interrupt Status
And
(set)
(reset)
Reg
MFC Data Storage Interrupt Status Register
MFC_DSISR[i]
Other MFC_DSISR bits
...
Or
(Level)
MFC LS Address Compare for put Detected
(Pulse)
MFC LS Address Compare for get Detected
(Pulse)
MFC Data-Segment Fault Detected
(Pulse)
Store Data[60]
Store Data[61]
Store Data[62]
Store Data[63]
And
And
And
And
(Set)
(Reset)
(Set)
(Reset)
(Set)
(Reset)
(Set)
(Reset)
Reg
Reg
Reg
Reg
Int_Stat_class1[LP]
Int_Stat_class1[LG]
Int_Stat_class1[MF]
Int_Stat_class1[SF]
Class 1 Interrupt Status Register
Int_Mask_class1[LP]
Int_Mask_class1[LG]
Int_Mask_class1MF]
Int_Mask_class1[SF]
And
And
And
And
Class 1 Interrupt Mask Register
Or
Generate interrupt packet if no packet for MFC Class 1 has been generated since the last PPE store to MFC Class 1 Interrupt Status

Note: It is undefined whether a “set” is set dominant or not, unless this is explicitly stated.

21.5.3 Class 2 Interrupts

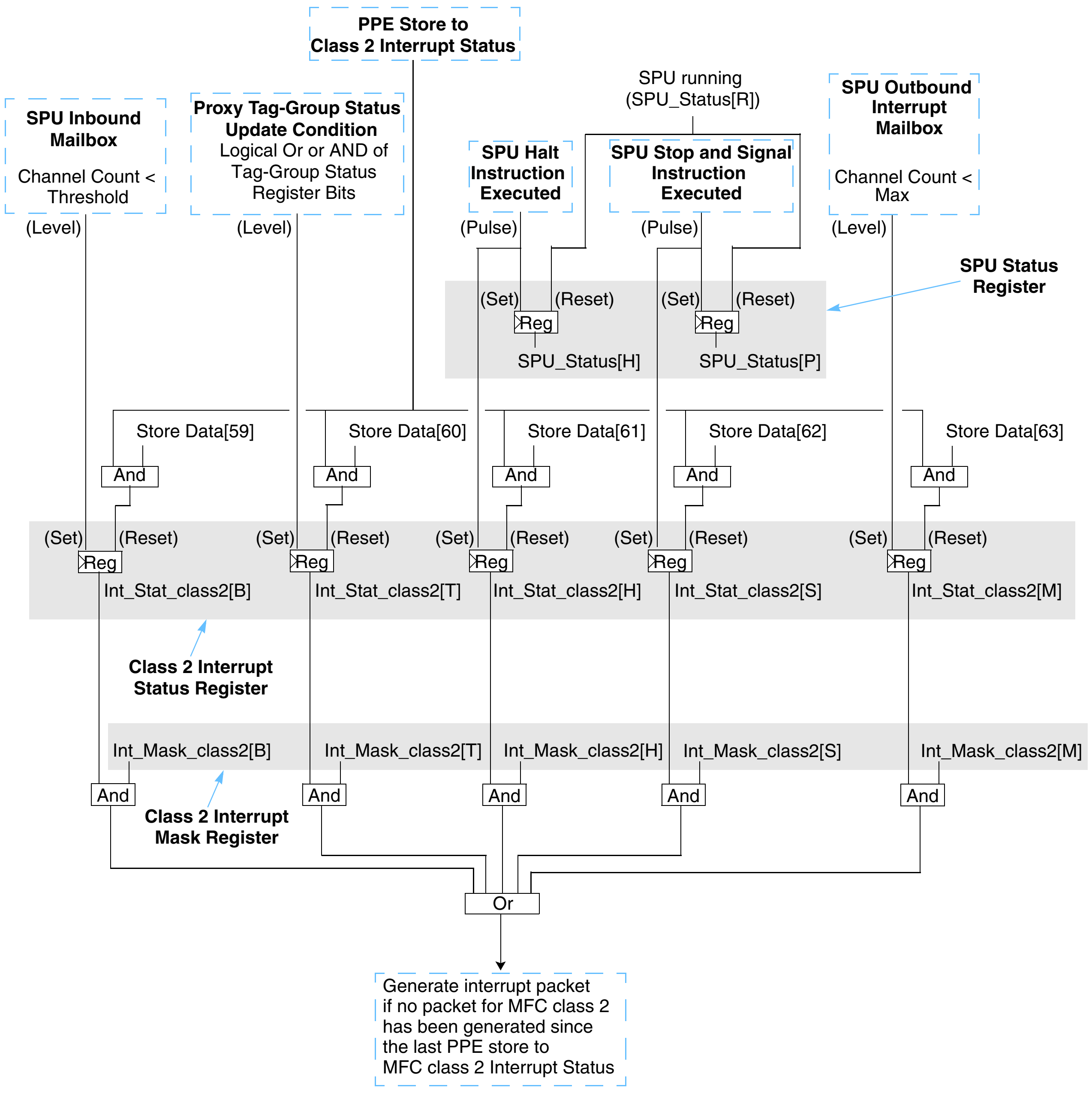
Several SPU application actions cause class 2 interrupts to be generated. *Figure 21-4* on page 275 illustrates the following process:

- If the amount of data in the SPU inbound mailbox queue falls below an implementation-dependent threshold, the B bit in the Class 2 Interrupt Status Register is set to '1'.
- If the Proxy Tag-Group Status Update condition is met, the T bit in the Class 2 Interrupt Status Register is set to '1'.
- If the SPU executes a halt instruction or a single instruction step completes, the H bit in the SPU Status Register is set to ‘1’, and the H bit in the Class 2 Interrupt Status Register is set to ‘1’.
- If the SPU executes a stop-and-signal instruction, the P bit in SPU_Status is set to ‘1’, and the S bit in the MFC Class 2 Interrupt Status Register is set to ‘1’.
- If the SPU writes to the SPU Outbound Interrupt Mailbox Register (see page 237), the SPU Write Outbound Interrupt Mailbox Channel count is decremented. Whenever the SPU Write Outbound Interrupt Mailbox Channel count is less than the maximum supported by the implementation, the M bit in the MFC Class 2 Interrupt Status Register is set to ‘1’.

Any enabled interrupts in the MFC Class 2 Interrupt Mask Register whose corresponding bit is set to ‘1’ in the MFC Class 2 Interrupt Status Register can cause an interrupt packet to be sent as described in *Section 21.8 Interrupt Routing Register* beginning on page 283.

The software interrupt handler can reset the class 2 MFC interrupt status bits without clearing SPU_Status. The SPU_Status bits are automatically reset when the SPU restarts.

Figure 21-4. MFC Class 2 Interrupt Generation Process


PPE Store to Class 2 Interrupt Status
SPU running (SPU_Status[R])
SPU Outbound Interrupt Mailbox
Channel Count < Max
SPU Inbound Mailbox
Channel Count < Threshold
Proxy Tag-Group Status Update Condition
Logical Or or AND of Tag-Group Status Register Bits
SPU Halt Instruction Executed
SPU Stop and Signal Instruction Executed
(Level)
(Level)
(Pulse)
(Pulse)
(Level)
SPU Status Register
(Set)
(Reset)
Reg
SPU_Status[H]
SPU_Status[P]
Store Data[59]
Store Data[60]
Store Data[61]
Store Data[62]
Store Data[63]
And
Int_Stat_class2[B]
Int_Stat_class2[T]
Int_Stat_class2[H]
Int_Stat_class2[S]
Int_Stat_class2[M]
Class 2 Interrupt Status Register
Int_Mask_class2[B]
Int_Mask_class2[T]
Int_Mask_class2[H]
Int_Mask_class2[S]
Int_Mask_class2[M]
Class 2 Interrupt Mask Register
Or
Generate interrupt packet if no packet for MFC class 2 has been generated since the last PPE store to MFC class 2 Interrupt Status

Note: It is undefined whether a “set” is set dominant or not, unless this is explicitly stated.

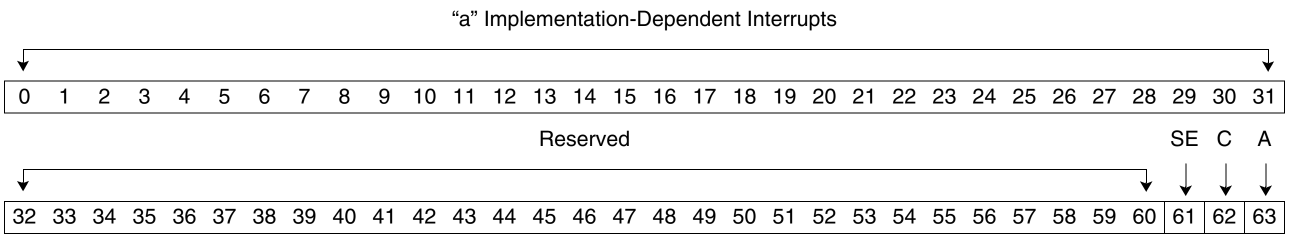
21.6 MFC Interrupt Mask Registers

There are three interrupt mask registers in each MFC: one for each class of interrupt (error, translation, application) as defined in *Section 21.4* on page 269. The interrupt mask registers allow privileged software to select which MFC and SPU events are allowed to generate an external interrupt to a PPE. Each bit in these registers has a corresponding status bit.

21.6.1 Class 0 Interrupt Mask Register (INT_Mask_class0)

Access Type Read/Write

Base Address Offset (BP_Base | P1(n)) + x‘0100’; where n is an SPE number.

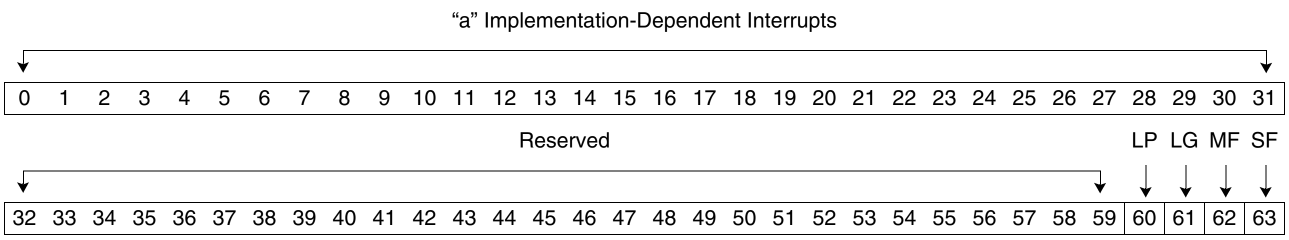


Bits	Field Name	Description
0:31	“a” Implementation-Dependent Interrupts	Mask for implementation-dependent interrupts. 0 Interrupt disabled. 1 Interrupt enabled.
32:60	Reserved	Reserved.
61	SE	Enable for SPU error interrupt. 0 Interrupt disabled. 1 Interrupt enabled.
62	C	Enable for invalid DMA command interrupt. 0 Interrupt disabled. 1 Interrupt enabled.
63	A	Enable for MFC DMA alignment interrupt. 0 Interrupt disabled. 1 Interrupt enabled.

21.6.2 Class 1 Interrupt Mask Register (INT_Mask_class1)

Access Type Read/Write

Base Address Offset (BP_Base | P1(n)) + x‘0108’; where n is an SPE number.

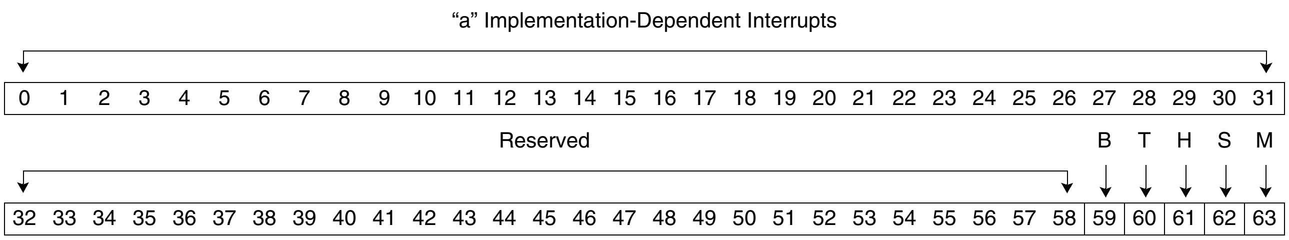


Bits	Field Name	Description
0:31	“a” Implementation-Dependent Interrupts	Mask for implementation-dependent interrupts. 0 Interrupt disabled. 1 Interrupt enabled.
32:59	Reserved	Reserved.
60	LP	Enable for MFC local storage address compare suspend on **put** interrupt. 0 Interrupt disabled. 1 Interrupt enabled.
61	LG	Enable for MFC local storage address compare suspend on **get** interrupt. 0 Interrupt disabled. 1 Interrupt enabled.
62	MF	Enable for MFC data-storage interrupt (mapping fault). 0 Interrupt disabled. 1 Interrupt enabled.
63	SF	Enable for MFC data segment interrupt (segment fault). 0 Interrupt disabled. 1 Interrupt enabled.

21.6.3 Class 2 Interrupt Mask Register (INT_Mask_class2)

Access Type Read/Write

Base Address Offset (BP_Base | P1(n)) + x‘0110’; where n is an SPE number.



Bits	Field Name	Description
0:31	“a” Implementation-Dependent Interrupts	Mask for implementation-dependent interrupts. 0 Interrupt disabled. 1 Interrupt enabled.
32:58	Reserved	Reserved.
59	B	Enable for SPU mailbox threshold interrupt. 0 Interrupt disabled. 1 Interrupt enabled.
60	T	Enable for DMA tag group completion. 0 Interrupt disabled. 1 Interrupt enabled.
61	H	Enable for SPU halt instruction trap or single instruction step complete. 0 Interrupt disabled. 1 Interrupt enabled.
62	S	Enable for SPU stop-and-signal instruction trap. 0 Interrupt disabled. 1 Interrupt enabled.
63	M	Enable for mailbox interrupt. 0 Interrupt disabled. 1 Interrupt enabled.

Note: Setting the threshold is implementation dependent.

21.7 MFC Interrupt Status Registers

There are three interrupt status registers for each MFC: one for each class of interrupt (error, translation, application) as defined in *Section 21.4* on page 269. The interrupt status registers allow privileged software to determine which MFC and SPU events caused the external interrupt to be presented to a PPE. See *PowerPC Architecture, Book III* for a description of how external interrupts are processed by a PPE. Each bit in the MFC Interrupt Status Registers has a corresponding mask in the MFC Interrupt Mask Registers defined in *Section 21.6* on page 276.

When an interrupt event occurs, the corresponding interrupt status bit is set. Reads of an Interrupt Status Register are nondestructive. To reset an interrupt status bit, software must write a ‘1’ to the bit corresponding to the interrupt status. Writing a ‘0’ to any bit location does not change the state of the interrupt status. If software writes a ‘1’ to an interrupt status bit at the same time that hardware sets the bit due to an interrupt event, the resulting value in the Interrupt Status Register is undefined if the condition that set the interrupt status bit is a pulse. This is the case for MFC class 0 interrupt status bits 61 through 63, MFC class 1 interrupt status bits 60, 61, and 63, and MFC class 2 interrupt status bits 61 and 62 (see *Figure 21-2 MFC Class 0 Interrupt Generation Process* on page 271, *Figure 21-3 MFC Class 1 Interrupt Generation Process* on page 273, and *Figure 21-4 MFC Class 2 Interrupt Generation Process* on page 275).

An interrupt packet is transmitted when the logical AND for the same class of the MFC Interrupt Status Register and the MFC Interrupt Mask Register is nonzero and an interrupt packet has not been transmitted since the last software write of the MFC Interrupt Status Register. Even though additional events can occur for a given class, only one interrupt packet for that class is sent until software stores to the MFC Interrupt Status Register for that class. After the store occurs, another interrupt packet is sent if the logical AND for the same class of the MFC Interrupt Status Register and the MFC Interrupt Mask Register is nonzero.

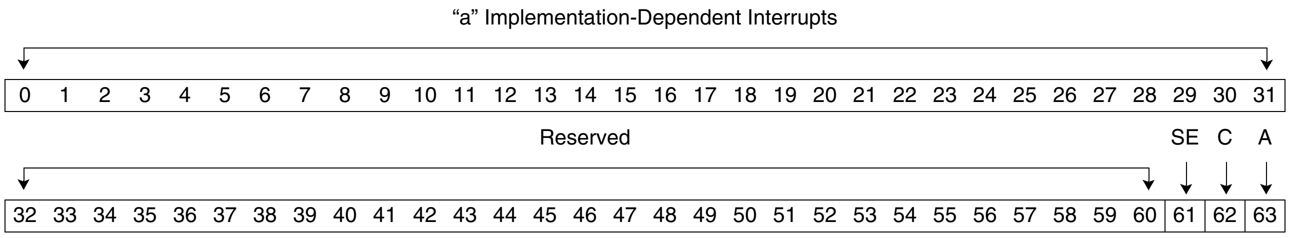
Implementation Note:

By the preceding definition, an interrupt must be generated if an interrupt event sets an MFC Interrupt Status Register bit to ‘1’ and the corresponding MFC Interrupt Mask Register bit is also ‘1’, provided an interrupt is not already pending for the specific interrupt class. In addition, an interrupt must be generated if an MFC Interrupt Status Register bit is ‘1’, and software sets the corresponding MFC Interrupt Mask Register bit to ‘1’, provided an interrupt is not already pending for the specific interrupt class.

21.7.1 Class 0 Interrupt Status Register (INT_Stat_class0)

Access Type Read/Write

Base Address Offset (BP_Base | P1(n)) + x‘0140’; where n is an SPE number.

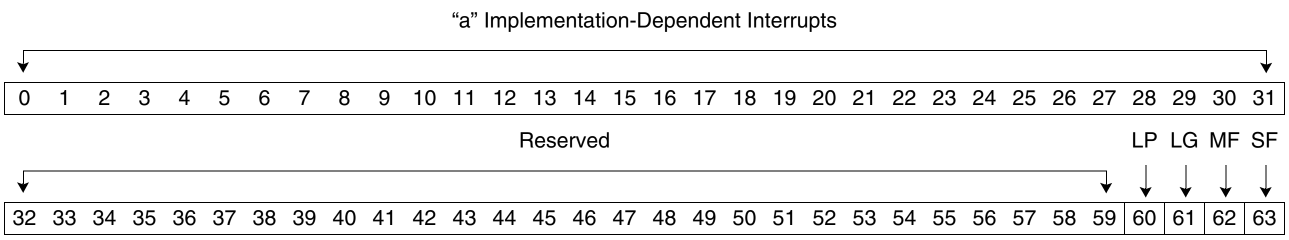


Bits	Field Name	Description
0:31	“a” Implementation-Dependent Interrupts	Status for implementation-dependent interrupts 0 Interrupt not pending for the corresponding interrupt type. 1 Interrupt pending for the corresponding interrupt type.
32:60	Reserved	Reserved
61	SE	Status for SPU error interrupt (see the specific implementation documentation for more details on errors) 0 Interrupt not pending for the corresponding interrupt type. 1 Interrupt pending for the corresponding interrupt type.
62	C	Status for invalid or illegal DMA command interrupt 0 Interrupt not pending for the corresponding interrupt type. 1 Interrupt pending for the corresponding interrupt type.
63	A	Status for DMA alignment interrupt 0 Interrupt not pending for the corresponding interrupt type. 1 Interrupt pending for the corresponding interrupt type.

21.7.2 Class 1 Interrupt Status Register (INT_Stat_class1)

Access Type Read/Write

Base Address Offset (BP_Base | P1(n)) + x‘0148’; where n is an SPE number.

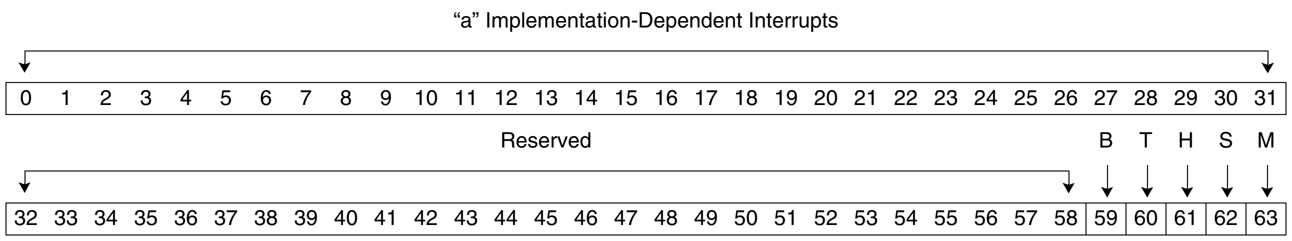


Bits	Field Name	Description
0:31	“a” Implementation-Dependent Interrupts	Status for implementation-dependent interrupts 0 Interrupt not pending for the corresponding interrupt type. 1 Interrupt pending for the corresponding interrupt type.
32:59	Reserved	Reserved
60	LP	Status for MFC local storage address compare suspend on **put** interrupt 0 Interrupt not pending for the corresponding interrupt type. 1 Interrupt pending for the corresponding interrupt type.
61	LG	Status for MFC local storage address compare suspend on **get** interrupt 0 Interrupt not pending for the corresponding interrupt type. 1 Interrupt pending for the corresponding interrupt type.
62	MF	Status for MFC data storage interrupt (mapping fault) 0 Interrupt not pending for the corresponding interrupt type. 1 Interrupt pending for the corresponding interrupt type.
63	SF	Status for MFC data segment interrupt (segment fault) 0 Interrupt not pending for the corresponding interrupt type. 1 Interrupt pending for the corresponding interrupt type.

21.7.3 Class 2 Interrupt Status Register (INT_Stat_class2)

Access Type Read/Write

Base Address Offset (BP_Base | P1(n)) + x'0150'; where n is an SPE number.



Bits	Field Name	Description
0:31	"a" Implementation-Dependent Interrupts	Status for implementation-dependent interrupts 0 Interrupt not pending for the corresponding interrupt type. 1 Interrupt pending for the corresponding interrupt type.
32:58	Reserved	Reserved
59	B	Status for SPU mailbox threshold interrupt 0 Interrupt not pending for the corresponding interrupt type. 1 Interrupt pending for the corresponding interrupt type.
60	T	Status for SPU tag-group complete interrupt 0 Interrupt not pending for the corresponding interrupt type. 1 Interrupt pending for the corresponding interrupt type.
61	H	Status for SPU halt instruction trap or single instruction step complete 0 Interrupt not pending for the corresponding interrupt type. 1 Interrupt pending for the corresponding interrupt type.
62	S	Status for SPU stop-and-signal instruction trap 0 Interrupt not pending for the corresponding interrupt type. 1 Interrupt pending for the corresponding interrupt type.
63	M	Status for mailbox interrupt 0 Interrupt not pending for the corresponding interrupt type. 1 Interrupt pending for the corresponding interrupt type.

21.8 Interrupt Routing Register (INT_Route)

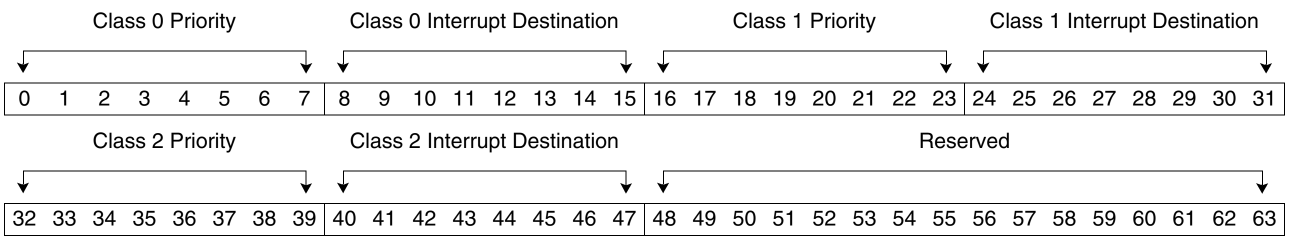
The Interrupt Routing Register allows privileged software to select which PPE or which external interrupt controller services to interrupt. The Interrupt Routing Register contains a set of routing information for each class of interrupt.

For each class, the register contains an 8-bit priority and an 8-bit interrupt destination. The interrupt destination indicates which logical PPE or external interrupt controller will be sent interrupt packets for MFC interrupts of the corresponding interrupt class.

The specific meaning of the Priority and Interrupt Destination fields is implementation dependent. See the specific implementation documentation for more information.

Access Type Read/Write

Base Address Offset (BP_Base | P1(n)) + x‘0180’; where n is an SPE number.



Bits	Field Name	Description
0:7	Class 0 Priority	Interrupt priority for class 0 interrupts. The value written to this field is used as the priority for all class 0 interrupts from the associated SPE.
8:15	Class 0 Interrupt Destination	Indicates which logical PPE or external interrupt controller is sent interrupt packets for MFC class 0 interrupts.
16:23	Class 1 Priority	Interrupt priority for class 1 interrupts. The value written to this field is used as the priority for all class 1 interrupts from the associated SPE.
24:31	Class 1 Interrupt Destination	Indicates which logical PPE or external interrupt controller is sent interrupt packets for MFC class 1 interrupts.
32:39	Class 2 Priority	Interrupt priority for class 2 interrupts. The value written to this field is used as the priority for all class 2 interrupts from the associated SPE.
40:47	Class 2 Interrupt Destination	Indicates which logical PPE or external interrupt controller is sent interrupt packets for MFC class 2 interrupts.
48:63	Reserved	Reserved

22. Power Management

The Cell Broadband Engine Architecture (CBEA) defines five major states for power management: Active, Slow (n), Pause (n), State Retained and Isolated (SRI), and State Lost and Isolated (SLI). Typically, the more aggressive the power management state, the more time is required to enter and exit the state (see *Table 22-1* to determine which states are more aggressive). These states apply to the various components of a CBEA-compliant processor as well as the full processor. The intent is to allow each component within a CBEA-compliant processor to be set to the same or different power-management states.

Multiple power-management states provide privileged software with the control necessary to manage the power of a CBEA-compliant processor while meeting the real-time demands of the system. Implementation of the various states, and the associated power savings, are implementation dependent. See the specific implementation documentation for more information.

Table 22-1. Power Management States

Power Management State	Power Savings	Description	Notes
Active	Less aggressive	In active state, the performance of a component is not limited by power management.	
Slow (n)	↓	In the slow (n) state, performance can be degraded for power savings. A higher value of n indicates a more aggressive power savings, and potential for more performance degradation. Real-time performance can be limited. The functionality of the component is not altered.	1
Pause (n)	↓	In the pause (n) state, the component is not guaranteed to make forward progress. The component state is maintained as well as the system integrity. A higher value of n indicates a more aggressive power savings, and potential for more performance degradation.	1
State Retained and Isolated	↓	In the 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.	
State Lost and Isolated	More aggressive	In the SLI state, the component is effectively removed from the system. The state of the component is not retained, and the component will not respond to a system event.	

1. Where *n* in the slow and pause states denotes the degree of power savings.

Typically, privileged software controls the transition between the various states. In some cases, an implementation can allow events to alter the power-management state of a CBEA-compliant processor.

23. Version Control

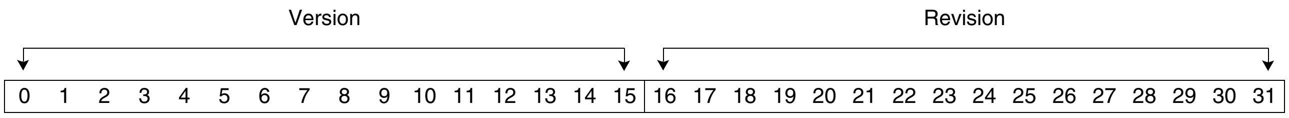
The Cell Broadband Engine Architecture (CBEA) consists of several components, each of which can have different version levels or revisions. To allow for this, the CBEA provides a version register for each component and an overall version register called the CBEA-Compliant Processor Version Register.

23.1 CBEA-Compliant Processor Version Register (BP_VR)

The CBEA-Compliant Processor Version Register is a 32-bit read only register that contains values that identify the version and revision level of the CBEA-compliant processor. The contents of the CBEA-Compliant Processor Version Register are only accessible using a PowerPC Processor Element (PPE) move from special-purpose register (**mfspr**) instruction. Read access to the CBEA-Compliant Processor Version Register is privileged. Write access is not provided. There is only one CBEA-Compliant Processor Version Register per CBEA-compliant processor.

Version numbers are assigned by the CBEA process. Revision numbers are assigned by an implementation-defined process.

Access Type	Read Only
PPE SPR Number	x‘3FE’



Bits	Field Name	Description
0:15	Version	A 16-bit number that identifies the version of the processor. Different version numbers indicate major differences between processors, such as which optional facilities and instructions are supported.
16:31	Revision	A 16-bit number that distinguishes between implementations of the version. Different revision numbers indicate minor differences between processors with the same version number, such as clock rate and engineering change level.

Implementation Note:

Although the classification of differences between CBEA-compliant processors as “major” or “minor” is somewhat arbitrary, the following are examples of differences that generally should be considered “major:”

- Number and type of execution units
- Optional facilities, instructions, and commands supported
- Level of instruction and command support (hard-wired or emulated)
- Size, geometry, and management of caches, translation lookaside buffers (TLBs), and synergistic processor unit (SPU) local storage

The following are examples of differences that generally should be considered “minor:”

- Remapping a processor to a new technology
- Redesigning a critical path to increase clock rate
- Fixing bugs

In general, any change to a CBEA-compliant processor should cause a new BP_VR value to be assigned. Any change to a CBEA-compliant processor, even a change that is not expected to be apparent to software, should cause a new revision number to be assigned in case the change has introduced an error that software must circumvent.

23.2 PPE Processor Version Register (PVR)

PowerPC Architecture, Book III describes a processor version register, which contains a 32-bit value that identifies the specific version (model) and revision level of the PPE portion of the CBEA.

There is one PVR for each PPE in a CBEA-compliant processor. The contents of the PPE Processor Version Register are only accessible using a PPE move from special-purpose register (**mfspr**) instruction.

Version numbers are assigned by the PowerPC Architecture process. Revision numbers are assigned by an implementation-defined process.

See *PowerPC Architecture, Book III* for a complete description of the PVR.

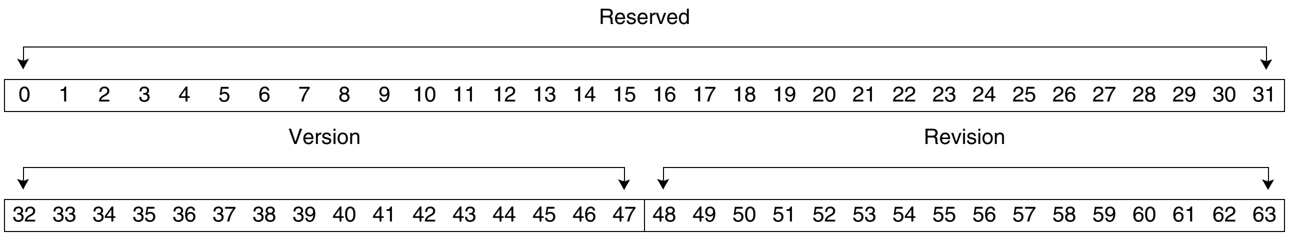
23.3 SPU Version Register (SPU_VR)

The SPU Version Register contains a 32-bit value that identifies the specific version (model) and revision level of the SPU portion of the CBEA. The contents of this register are accessible from a PPE using a load doubleword (**ld**) instruction. Read access to the SPU Version Register should be privileged. Write access is not provided. Access to this register from the SPU is not provided. There is one SPU Version Register for each SPU in a CBEA-compliant processor.

The SPU_VR distinguishes between SPUs that differ in attributes that can affect software. It contains two fields. Version numbers are assigned by the SPU Architecture process. Revision numbers are assigned by an implementation-defined process.

Access Type Read Only

Base Address Offset (BP_Base | P1(n)) + x‘0020’, where n is an SPE number.



Bits	Field Name	Description
0:31	Reserved	Set to zeros.
32:47	Version	A 16-bit number that identifies the version of the SPU. Different version numbers indicate major differences between SPUs, such as which optional facilities and instructions are supported.
48:63	Revision	A 16-bit number that distinguishes between implementations of the version. Different revision numbers indicate minor differences between SPUs with the same version number, such as clock rate and engineering change level.

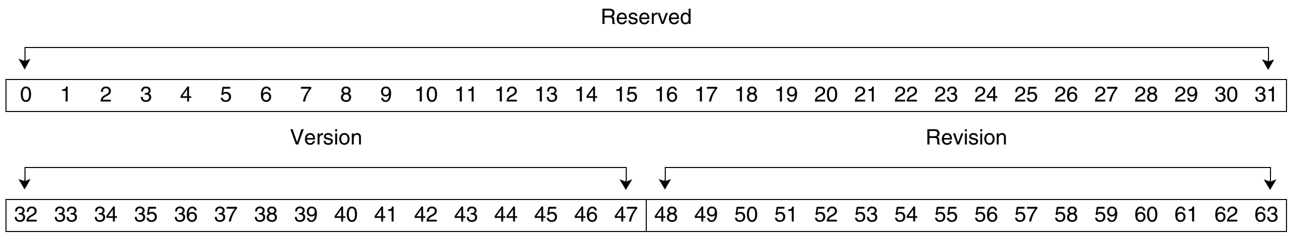
23.4 MFC Version Register (MFC_VR)

The MFC Version Register contains a 32-bit value that identifies the version and revision level of the memory flow controller (MFC) component of the CBEA. The contents of the MFC Version Register are accessible from a PPE using a **ld** instruction. Read access to the MFC_VR should be privileged. Write access is not provided. There is one MFC Version Register for each MFC in the CBEA-compliant processor.

The MFC_VR distinguishes between MFCs that differ in attributes that can affect software. It contains two fields. Version numbers are assigned by the CBEA process. Revision numbers are assigned by an implementation-defined process.

Access Type Read Only

Base Address Offset (BP_Base | P1(n)) + x‘0018’, where n is an SPE number.



Bits	Field Name	Description
0:31	Reserved	Set to zeros.
32:47	Version	A 16-bit number that identifies the version of the MFC. Different version numbers indicate major differences between MFC units, such as which optional facilities and instructions are supported.
48:63	Revision	A 16-bit number that distinguishes between implementations of the version. Different revision numbers indicate minor differences between MFC units with the same version number, such as clock rate and engineering change level.

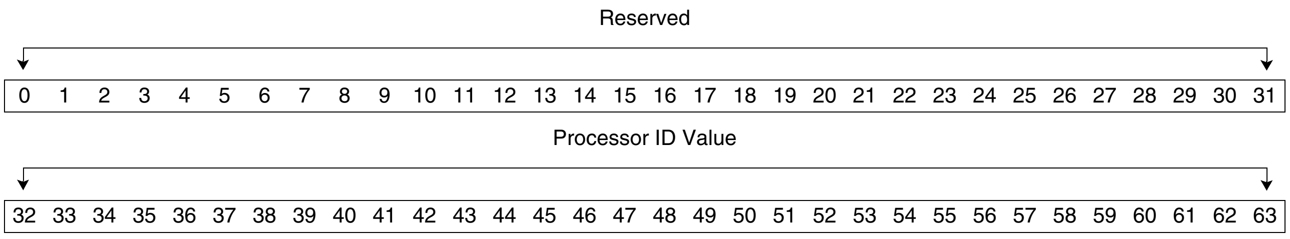
23.5 SPU Identification Register (SPU_ID)

The SPU Identification Register contains a 32-bit value that can be used to distinguish an SPU from other SPUs in the system. The contents of this register are accessible from a PPE using an **ld** instruction. Read access to the SPU Identification Register should be privileged. Write access, if provided, is implementation dependent. (See the specific implementation documentation for more information.)

Access to this register from the SPU is not provided. There is one SPU Identification Register for each SPU in the CBEA-compliant processor.

SPU Identification Register initialization is implementation dependent. See the specific implementation documentation for more information.

Access Type	Read/Write
Base Address Offset	(BP_Base I P1(n)) + x‘0010’, where n is an SPE number.



Bits	Field Name	Description
0:31	Reserved	Set to zeros.
32:63	Processor ID Value	Distinguishes the SPU from other SPUs in the system.

Appendix A. Memory Maps

This appendix contains the mapping of all registers defined by the Cell Broadband Engine Architecture (CBEA) in the real address space. As shown in *Table A-1 CBEA-Compliant Processor Memory Map* on page 294, the memory map for a single CBEA-compliant processor is divided into six sections:

- Synergistic Processor Element (SPE) Local Storage (LS), Problem State (PS), and Privilege 2 (P2) Area
- SPE Privilege 1 Area
- PowerPC Processor Element (PPE) Area
- Internal Interrupt Controller (IIC) Area
- Power Management and Debug (PMD) Area
- Implementation-Dependent Expansion Area (IDEA)

The size of each area is based on the number of units (PPEs, SPUs, and IICs) in the CBEA-compliant processor.

The starting location for the memory map is defined in an implementation-dependent base address register (BP_Base). The BP_Base is used to calculate the address for all registers. An implementation can place address holes in the memory map (that is, areas where registers are not defined) to simplify decoding of the registers. BP_Base for a CBEA-compliant processor is implementation dependent. See the specific implementation documentation to determine how the base registers are initialized.

Implementation Note:

A CBEA-compliant processor implementation must provide at least one base register for relocating the internal registers. All base register (BP_Base) values must be initialized during the CBEA-compliant processor initialization sequence. The base register can either be set using a program sequence or initialized using a hardware method such as scan or JTAG.

Table A-1. CBEA-Compliant Processor Memory Map (Page 1 of 2)

Real Address (Hexadecimal)	Area	Description
SPE Local Storage (LS(n)), Problem State (PS(n)), and Privilege 2 (P2(n)) Area; where $0 \le n \le$ (number of SPEs -1)[1] • LS(0) = BP_Base:[2] The starting address of the area. • spe_area_size = (2 × LS_size) rounded up to a power of 2 boundary.		
Local Storage Area for SPE(n); where LS(n) = n × spe_area_size		
BP_Base | LS(n)	Start of LS(n) area	Start of the local storage area for SPE(n)
BP_Base | LS(n) + LS_Size - 1	End of LS(n) area	End of the local storage area
Problem State Area for SPE(n); where PS(n) = LS(n) + (spe_area_size >> 1) [3]		
BP_Base | PS(n)	Start of PS(n) area	Start of the problem state area for SPE(n)
BP_Base | PS(n) + x'1FFFF'	End of PS(n) area	End of the problem state area
Privilege 2 Area for SPE(n); where P2(n) = PS(n) + x'20000'		
BP_Base | P2(n)	Start of P2(n) area	Start of the privilege 2 area for SPE(n)
BP_Base | P2(n) + x'1FFFF'	End of P2(n) area	End of the privilege 2 area
SPE Privilege 1 Area (P1(n)); where $0 \le n <$ number of SPEs [1] • P1(0) ≥ P2(max) + x'20000': The starting address of the area. • P1(n) = P1(0) + (n × x'2000'); where $0 \le n \le$ (number of SPEs - 1).		
BP_Base | P1(n)	Start of P1(n) area	Start of the privilege 1 area for SPE(n)
BP_Base | P1(n) + x'1FFF'	End of P1(n) area	End of the privilege 1 area

Notes:

1. The value "n" ranges from zero to the number of SPEs minus 1 ($0 \le n \le$ (number of SPEs - 1)). If the number of SPEs is not a power of 2, an implementation can choose to increase the value of "n" to the next power of 2 boundary and reserve the extra space. Doing so simplifies decoding the address ranges.
2. This table assumes that the base starts on a power of 2 boundary that is greater than or equal to the size of the memory map area.
3. The symbol >> indicates shift by one to the right.
4. The value "p" ranges from zero to the number of PPEs (not threads) minus 1 ($0 \le p \le$ [number of PPEs - 1]). If the number of PPEs is not a power of 2, an implementation can choose to increase the value of "p" to the next power of 2 boundary and reserve the extra space. Doing so simplifies decoding the address ranges.
5. The value "t" ranges from zero to the number of PPU threads in the associated PPE minus 1 ($0 \le t \le$ [number of PPU threads - 1]).
6. The memory map pad (MM_pad) is used to pad the CBEA-compliant processor memory-mapped area to at least a 64 KB boundary.

Table A-1. CBEA-Compliant Processor Memory Map (Page 2 of 2)

Real Address (Hexadecimal)	Area	Description
PPE Area(p); where 0 ≤ p ≤ number of physical PPEs - 1 (*PPEs*; not *PPE* threads)[4] • PPE(0) ≥ P1(max) + x'2000': the starting address of the area. • PPE(p) = PPE(0) + (p × x'1000'); where 0 ≤ n ≤ (number of PPEs - 1).		
BP_Base I PPE(p)	Start of PPE(p) area	Start of the privilege 1 area for PPE(p)
BP_Base I PPE(p) + x'FFF'	End of PPE(p) area	End of the PPE(p) area
Internal Interrupt Controller Area(p); where 0 ≤ p ≤ number of physical PPEs - 1 (*PPEs*; not PPE *threads*)[4] • IIC(0) ≥ PPE(max) + x'1000': the starting address of the area. • IIC(p) = IIC(0) + (p × x'1000'). **Note:** If more than 16 threads per PPE are needed, additional IIC areas are added, and the threads are divided between the areas.		
BP_Base I IIC(p)	Start of IIC(p) implementation-dependent area	Start of the implementation-dependent area for the IIC
BP_Base I IIC(p) + x'3FF'	End of IIC(p) implementation-dependent area	End of the implementation-dependent area for the IIC
Internal Interrupt Controller Thread Control Block(t); where 0 ≤ t ≤ number of PPU threads - 1[5] • IIC_TCB(t) = IIC(p) + x'400' + (t × x'20')		
BP_Base I IIC_TCB(t)	Start of IIC_TCB(t) area	Start of the thread control block area for IIC_TCB(t) (see *Section 21.3 Internal Interrupt Controller Registers* on page 263)
BP_Base I IIC_TCB(t) + x'1F'	End of IIC_TCB(t) area	End of the thread control block area for IIC_TCB(t)
BP_Base I IIC(p) + x'FFF'	End of IIC(p) area	End of the internal interrupt controller area
Power Management and Debug Area • PMD ≥ IIC(max) + x'1000': The starting address of the area.		
BP_Base I PMD	Start of PMD area	Start of the memory-mapped I/O (MMIO) area for power management and architected performance monitor and debug features
BP_Base I PMD + x'FFF'	End of PMD area	End of the power management and architected performance monitor and debug area
Implementation-Dependent Expansion Area • IDEA ≥ PMD + x'1000': The starting address of the area.		
BP_Base I IDEA	Start of the IDEA area	Start of the MMIO area for implementation-dependent features defined in the specific implementation documentation
BP_Base I IDEA + MM_pad[6]	End of the IDEA area	End of the MMIO area for implementation-dependent features

Notes:
1. The value "n" ranges from zero to the number of SPEs minus 1 (0 ≤ n ≤ (number of SPEs - 1)). If the number of SPEs is not a power of 2, an implementation can choose to increase the value of "n" to the next power of 2 boundary and reserve the extra space. Doing so simplifies decoding the address ranges.
2. This table assumes that the base starts on a power of 2 boundary that is greater than or equal to the size of the memory map area.
3. The symbol >> indicates shift by one to the right.
4. The value "p" ranges from zero to the number of PPEs (not threads) minus 1 (0 ≤ p ≤ [number of PPEs - 1]). If the number of PPEs is not a power of 2, an implementation can choose to increase the value of "p" to the next power of 2 boundary and reserve the extra space. Doing so simplifies decoding the address ranges.
5. The value "t" ranges from zero to the number of PPU threads in the associated PPE minus 1 (0 ≤ t ≤ [number of PPU threads - 1]).
6. The memory map pad (MM_pad) is used to pad the CBEA-compliant processor memory-mapped area to at least a 64 KB boundary.

A.1 SPE Problem State Memory Map

Table A-2 shows how the SPU problem state registers are mapped into the real address space of the system. Some registers are defined as byte and half-word widths, but all accesses to these registers are required to be a minimum of 32 bits.

Table A-2. SPE Problem State Memory Map (Page 1 of 2)

Offset (Hexadecimal)	Register	Description	Access Type
Multisource Synchronization Area			
x‘00000’	MFC_MSSync	MFC Multisource Synchronization Register (see page 109)	Read/Write
Memory Flow Controller (MFC) Proxy Command Parameter Area			
x‘03000’	Reserved	Reserved for future expansion.	Reserved
x‘03004’	MFC_LSA	MFC Local Storage Address Register (see page 85)[1]	Write Only
x‘03008’	MFC_EAH	MFC Effective Address High Register (see page 86)[1]	Write Only
x‘0300C’	MFC_EAL	MFC Effective Address Low Register (see page 87)[1]	Write Only
x‘03010’	MFC_Size	MFC Transfer Size Register (see page 84)[1, 2] (Upper 16 bits of register)	Write Only
	MFC_Tag	MFC Command Tag Register (see page 83)[1, 2] (Lower 16 bits of register)	Write Only
x‘03014’	MFC_ClassID	MFC Class ID Register (see page 82)[1, 3] (Upper 16 bits of register for write)	Write Only
	MFC_Cmd	MFC Command Opcode Register (see page 81) [1, 3] (Lower 16 bits of register for write)	Write Only
	MFC_CMDStatus	MFC Command Status Register (see page 90)	Read Only
MFC Proxy Status and Command Queue Control Area			
x‘03020’:x‘030FF’	Reserved	Reserved	
x‘03104’	MFC_QStatus	MFC Queue Status Register (see page 91)	Read Only
x‘03204’	Prxy_QueryType	Proxy Tag-Group Query Type Register (see page 93)	Read/Write
x‘0321C’	Prxy_QueryMask	Proxy Tag-Group Query Mask Register (see page 94)	Read/Write
x‘0322C’	Prxy_TagStatus	Proxy Tag-Group Status Register (see page 95)	Read Only
x‘03330’:x‘03FFF’	Reserved	Reserved	
Synergistic Processor Unit (SPU) Control Area			
x‘04004’	SPU_Out_Mbox	SPU Outbound Mailbox Register (see page 102)	Read Only
x‘0400C’	SPU_In_Mbox	SPU Inbound Mailbox Register (see page 103)[1]	Write Only
x‘04014’	SPU_Mbox_Stat	SPU Mailbox Status Register (see page 104)	Read Only
x‘0401C’	SPU_RunCntl	SPU Run Control Register (see page 96)	Read/Write
x‘04024’	SPU_Status	SPU Status Register (see page 97)	Read Only
x‘04034’	SPU_NPC	SPU Next Program Counter Register (see page 99)	Read/Write
x‘04038’:x‘13FFF’	Reserved	Reserved	

1. Reading of these registers should be allowed for diagnostic purposes.
2. Both the MFC_Size and MFC_TAG registers must be written with a single 32-bit store instruction.
3. Both the MFC_ClassID and MFC_Cmd registers must be written with a single 32-bit store instruction.

Table A-2. SPE Problem State Memory Map (Page 2 of 2)

Offset (Hexadecimal)	Register	Description	Access Type
Signal-Notification Area			
x‘1400C’	SPU_Sig_Notify_1	SPU Signal Notification 1 Register (see page 106)	Read/Write
x‘14010’:x‘1BFFF’	Reserved	Reserved	
x‘1C00C’	SPU_Sig_Notify_2	SPU Signal Notification 2 Register (see page 107)	Read/Write
x‘1C010’:x‘1FFFF’	Reserved	Reserved	

1. Reading of these registers should be allowed for diagnostic purposes.
2. Both the MFC_Size and MFC_TAG registers must be written with a single 32-bit store instruction.
3. Both the MFC_ClassID and MFC_Cmd registers must be written with a single 32-bit store instruction.

A.2 SPE Privilege 1 Memory Map

Table A-3 lists all the CBEA-compliant SPE registers that allow only privilege 1 access.

Table A-3. SPE Privilege 1 Memory Map (Page 1 of 3)

Offset (Hexadecimal)	Register	Description	Access Type
Control and Configuration Area			
x‘0000’	MFC_SR1	MFC State Register One (see page 221)	Read/Write
x‘0008’	MFC_LPID	MFC Logical Partition ID Register (see page 223)	Read/Write
x‘0010’	SPU_ID	SPU Identification Register (see page 291)	Read/Write
x‘0018’	MFC_VR	MFC Version Register (see page 290)	Read Only
x‘0020’	SPU_VR	SPU Version Register (see page 289)	Read Only
x‘0028’:x‘00FF’	Reserved	Reserved	
Interrupt Area			
x‘0100’	INT_Mask_class0	Class 0 Interrupt Mask Register (see page 276)	Read/Write
x‘0108’	INT_Mask_class1	Class 1 Interrupt Mask Register (see page 277)	Read/Write
x‘0110’	INT_Mask_class2	Class 2 Interrupt Mask Register (see page 278)	Read/Write
x‘0118’:x‘013F’	Reserved	Reserved	Reserved
x‘0140’	INT_Stat_class0	Class 0 Interrupt Status Register (see page 280)	Read/Write
x‘0148’	INT_Stat_class1	Class 1 Interrupt Status Register (see page 281)	Read/Write
x‘0150’	INT_Stat_class2	Class 2 Interrupt Status Register (see page 282)	Read/Write
x‘0158’:x‘017F’	Reserved	Reserved	Reserved
x‘0180’	INT_Route	Interrupt Routing Register (see page 283)	Read/Write
x‘0188’:x‘01FF’	Reserved	Reserved	Reserved
Atomic Unit Control Area			
x‘0200’	MFC_Atomic_Flush	MFC Atomic Flush Register (see page 236) This is an implementation-dependent register	Read/Write
x‘0208’:x‘03FF’	SPU_Cache_ImplRegs	SPU cache hardware implementation-dependent registers. See the specific implementation documentation.	

1. An implementation should support reading of these registers for diagnostic purposes.

Table A-3. SPE Privilege 1 Memory Map (Page 2 of 3)

Offset (Hexadecimal)	Register	Description	Access Type
Translation Lookaside Buffer (TLB) Management Registers			
x‘0400’	MFC_SDR	MFC Storage Description Register (see page 224) See also the *PowerPC Architecture, Book III* for a description of this register.	Read/Write
x‘0408’:x‘04FF’	Reserved	Reserved	
x‘0500’	MFC_TLB_Index_Hint	TLB Index Hint Register (see page 209) Index to the best TLB entry to update.	Read Only
x‘0508’	MFC_TLB_Index	TLB Index Register (see page 210)[1] Index to the TLB entry to update with TLB Real Page Number Register and TLB Virtual Page Number Register.	Write Only
x‘0510’	MFC_TLB_VPN	TLB Virtual Page Number Register (see page 211) Access to the upper portion of the TLB entry.	Read/Write
x‘0518’	MFC_TLB_RPN	TLB Real Page Number Register (see page 212) Access to the lower portion of the TLB entry.	Read/Write
x‘0520’:x‘053F’	Reserved	Reserved	
x‘0540’	MFC_TLB_Invalidate_Entry	TLB Invalidate Entry Register (see page 214)[1] Virtual page number of the TLB entry to invalidate. **Note:** Not available for a PPE.	Write Only
x‘0548’	MFC_TLB_Invalidate_All	TLB Invalidate All Register (see page 216)[1] Invalidate all TLB entries (optional). **Note:** Not available for a PPE.	Write Only
x'0550':'057F’	Reserved	Reserved	
Memory Management. (Implementation-dependent area. See the specific implementation documentation.)			
x‘0580’:x‘05FF’	SPE_MMU_ImplRegs	SPE Memory Management Unit (MMU) Registers See the specific implementation documentation for a description of this register.	
MFC Status and Control Area			
x‘0600’	MFC_ACCR	MFC Address Compare Control Register (see page 227)	Read/Write
x‘0610’	MFC_DSISR	MFC Data Storage Interrupt Status Register (see page 226)	Read/Write
x‘0620’	MFC_DAR	MFC Data Address Register (see page 225)	Read/Write
x‘0628’:x‘06FF’	Reserved	Reserved	
Replacement Management Table Area (RMT) (Implementation-dependent area. See the specific implementation documentation).			
x‘0700’	MFC_TLB_RMT_Index	RMT Index Register (see page 257) Index to the replacement management tables.	Read/Write
x‘0710’	MFC_TLB_RMT_Data	RMT Data Register (see page 258) Doubleword of RMT data pointed to by the RMT Index Register. Entry contents are implementation dependent.	Read/Write
x‘0718’:x‘07FF’	SPE_RMT_ImplRegs	SPE RMT hardware implementation-dependent registers	

1. An implementation should support reading of these registers for diagnostic purposes.

Table A-3. SPE Privilege 1 Memory Map (Page 3 of 3)

Offset (Hexadecimal)	Register	Description	Access Type
MFC Command Data Storage Interrupt Area			
x‘0800’	MFC_DSIPR	MFC Data Storage Interrupt Pointer Register (see page 232) Contains a pointer to the command in the MFC command queue that caused the error condition.	Read Only
x‘0808’	MFC_LSACR	MFC Local Storage Address Compare Register (see page 229) 64-bit MFC Local Storage Address Compare Register	Read/Write
x‘0810’	MFC_LSCRR	MFC Local Storage Compare Result Register (see page 230) 64-bit MFC Local Storage Compare Results Register	Read Only
x‘0818’:x‘08FF’	Reserved	Reserved	
Real-Mode Support Registers			
x‘0900’	MFC_RMAB	MFC Real-Mode Address Boundary Register (see page 218)	Read/Write
x‘0908’:x‘0BFF’	Reserved	Reserved	
MFC Command Error Area			
x‘0C00’	MFC_CER	MFC Command Error Register (see page 231) Contains a pointer to the command in the direct memory access (DMA) queue that caused the error condition.	Read Only
x‘0C08’:x‘0FFF’	Reserved	Reserved	
Implementation-Dependent Area. (See the specific implementation documentation for a detailed description of these registers).			
x‘1000’:x‘1FFF’	PV1_ImplRegs	Privilege 1 implementation-dependent registers	

1. An implementation should support reading of these registers for diagnostic purposes.

A.3 SPE Privilege 2 Memory Map

Table A-4 lists all the CBEA-compliant registers that allow only privilege 2 access.

Table A-4. SPE Privilege 2 Memory Map (Page 1 of 2)

Offset (Hexadecimal)	Register	Description	Access Type
MFC Registers			
x'00000':x'010FF'	Reserved	Reserved	Reserved
Segment Lookaside Buffer Management Registers			
x'01100'	Reserved	Reserved	Reserved
x'01108'	SLB_Index	SLB Index Register (see page 201)[1] Index to the segment lookaside buffer (SLB) entry to be updated by the SLB_VSID and SLB_ESID ports.	Write Only
x'01110'	SLB_ESID	SLB Effective Segment ID Register (see page 202) Access to the upper portion of an SLB entry.	Read/Write
x'01118'	SLB_VSID	SLB Virtual Segment ID Register (see page 203) Access to the lower portion of an SLB entry.	Read/Write
x'01120'	SLB_Invalidate_Entry	SLB Invalidate Entry Register (see page 205)[1] Effective segment ID (ESID) of the SLB entry to invalidate.	Write Only
x'01128'	SLB_Invalidate_All	SLB Invalidate All Register (see page 206)[1] Invalidate all SLB entries.	Write Only
x'01130':x'01FFF'	Reserved	Reserved	Reserved
Context Save-and-Restore Area (Implementation-Dependent Area. See the specific implementation documentation.)			
x'02000':x'02FFF'	MFC_CSR_ImplRegs	MFC Context Save and Restore registers	
MFC Control			
x'03000'	MFC_CNTL	MFC Control Register (see page 233)	Read/Write
x'03008':x'03FFFF'	MFC_Cntl1_ImplRegs	Implementation-dependent control registers. See the specific implementation documentation.	
Interrupt Mailbox			
x'04000'	SPU_Out_Intr_Mbox	SPU Outbound Interrupt Mailbox Register (see page 237) SPU writes; PPE reads.	Read Only
SPU Control			
x'04040'	SPU_PrivCntl	SPU Privileged Control Register (see page 239)	Read/Write
x'04058'	SPU_LSLR	SPU Local Storage Limit Register (see page 241)	Read/Write
x'04060'	SPU_ChnlIndex	SPU Channel Index Register (see page 242) This register selects which SPU channel in the specified SPU(n) is accessed using the SPU Channel Count Register or SPU Channel Data Register.	Read/Write
x'04068'	SPU_ChnlCnt	SPU Channel Count Register (see page 244) This register reads or initializes the SPU Channel Count Register selected by the SPU Channel Index Register.	Read/Write

1. An implementation should support reading of these registers for diagnostic purposes.

Table A-4. SPE Privilege 2 Memory Map (Page 2 of 2)

Offset (Hexadecimal)	Register	Description	Access Type
x‘04070’	SPU_ChnlData	SPU Channel Data Register (see page 243) This register reads or initializes the SPU channel data selected by the SPU Channel Index Register.	Read/Write
x‘04078’	SPU_Cfg	SPU Configuration Register (see page 245) This register reads or sets the configuration of the SPU Signal-Notification Registers in the specified SPU(n).	Read/Write
x‘04080’:x‘04FFF’	Reserved	Reserved	
Implementation-Dependent Area. (See the specific implementation documentation for a detailed description of these registers.)			
x‘05000’:x‘0FFFF’	PV2_ImplRegs	Privilege 2 implementation-dependent registers	
Reserved Area			
x‘10000’:x‘1FFFF’	Reserved	Reserved	

1. An implementation should support reading of these registers for diagnostic purposes.

A.4 PPE Privilege 1 Memory Map

Table A-5 lists all the CBEA-compliant PPE registers that require privilege 1 access.

Table A-5. PPE Privilege 1 Memory Map

Offset (Hexadecimal)	Register	Description	Access Type
RMT Area (Implementation-Dependent Area. See the specific implementation documentation.)			
x‘000’:x‘2FF’	Reserved	Reserved	
x‘300’	L2_RMT_Index	RMT Index Register (see page 257). Index to the replacement-management table.	Read/Write
x‘310’	L2_RMT_Data	RMT Data Register (see page 258). Doubleword of RMT data pointed to by the RMT Index Register. Entry contents are implementation dependent.	Read/Write
x‘318’:x‘7FF’	Reserved	Reserved	
Implementation-Dependent Area (See the specific implementation documentation for a detailed description of these registers)			
x‘800’:x‘FFF’	PUPV_ImplRegs	Implementation-dependent PPE privileged-state registers.	

A.5 Internal Interrupt Controller Memory Map

The IIC has an interrupt control block for each physical and logical PPE (that is, for each thread in the physical processor). These control blocks are mapped in the real address space, starting at an implementation-dependent offset from the BP_Base. The control block's starting address is defined as the BP_Base | IIC(p) + x'400' + (t × x'20'); where 'p' is the physical PPE and 't' is the PPU thread number for the corresponding PPE.

(See *Table A-1 CBEA-Compliant Processor Memory Map* on page 294 for more details.)

Table A-6 shows the registers associated with each control block and their offsets from starting address of the BP_Base.

Table A-6. Internal Interrupt Controller Memory Map

Offset (Hexadecimal)	Register	Description	Access Type
PowerPC Processor Unit (PPU) (Thread t) Interrupt Control Block where 0 ≤ t ≤ number of PPU threads -1			
x'000'	INT_Pending_NonD	Interrupt Pending Port Registers (see page 263), nondestructive read. Status and data for pending interrupt.	Read Only
x'008'	INT_Pending_D	Interrupt Pending Port Registers (see page 263), destructive read. Status and data for pending interrupt.	Read Only
x'010'	INT_Generation	Interrupt Generation Port Register (see page 267). Port for the generation of an interprocessor interrupt (IPI).	Write Only
x'018'	INT_CPL	Interrupt Current Priority Level Register (see page 268). Only higher-priority interrupts cause an external interrupt.	Read/Write

Appendix B. SPU Channel Map

This appendix contains the mapping of all channels defined by the Cell Broadband Engine Architecture (CBEA) in the real address space. The channel map for a single SPU in a CBEA-compliant processor is divided into several sections:

- Synergistic Processor Unit (SPU) Event Channels
- SPU Signal Notification Channels
- SPU Decrementer Channels
- SPU Multisource Synchronization Channel
- Mask Read Channels
- SPU State Management Channels
- Memory Flow Controller (MFC) Command Parameter Channels
- MFC Tag Status Channels
- SPU Mailbox Channels

Note: No reserved channels can be used for implementation-dependent functions.

Table B-1. SPU Channel Map (Page 1 of 3)

Channel Number (Hexadecimal)	Channel Name	Description	Access Type
SPU Event Channels			
x‘0’	SPU_RdEventStat	SPU Read Event Status Channel (see page 153). Read event status (with mask applied).	Read blocking
x‘1’	SPU_WrEventMask	SPU Write Event Mask Channel (see page 157). Write event-status mask.	Write
x‘2’	SPU_WrEventAck	SPU Write Event Acknowledgment Channel (see page 161). Write end-of-event processing.	Write
SPU Signal Notification Channels			
x‘3’	SPU_RdSigNotify1	SPU Signal Notification 1 Channel (see page 143).	Read blocking
x‘4’	SPU_RdSigNotify2	SPU Signal Notification 2 Channel (see page 144).	Read blocking
x‘5’	Channel 5	Reserved	
x‘6’	Channel 6	Reserved	
SPU Decrementer Channels			
x‘7’	SPU_WrDec	SPU Write Decrementer Channel (see page 145).	Write
x‘8’	SPU_RdDec	SPU Read Decrementer Channel (see page 146).	Read
MFC Multisource Synchronization Channels			
x‘9’	MFC_WrMSSyncReq	MFC Write Multisource Synchronization Request Channel (see page 149).	Write blocking
SPU Reserved Channel			
x‘A’	Channel 10	Reserved	
Mask Read Channels			
x‘B’	SPU_RdEventMask	SPU Read Event Mask Channel (see page 159).	Read

Table B-1. SPU Channel Map (Page 2 of 3)

Channel Number (Hexadecimal)	Channel Name	Description	Access Type
x‘C’	MFC_RdTagMask	MFC Read Tag-Group Query Mask Channel (see page 131).	Read
SPU State Management Channels			
x‘D’	SPU_RdMachStat	SPU Read Machine Status Channel (see page 147).	Read
x‘E’	SPU_WrSRR0	SPU Write State Save-and-Restore Channel (see page 148).	Write
x‘F’	SPU_RdSRR0	SPU Read State Save-and-Restore Channel (see page 148).	Read
MFC SPU Command Parameter Channels			
x‘10’	MFC_LSA	MFC Local Storage Address Channel (see page 121). Write local storage address command parameter.	Write
x‘11’	MFC_EAH	MFC Effective Address High Channel (see page 124). Write high-order MFC effective-address command parameter.	Write
x‘12’	MFC_EAL	MFC Effective Address Low or List Address Channel (see page 122). Write low-order MFC effective-address command parameter.	Write
x‘13’	MFC_Size	MFC Transfer Size or List Size Channel (see page 120). Write MFC transfer-size command parameter.	Write
x‘14’	MFC_TagID	MFC Command Tag Identification Channel (see page 119). Write Tag identifier command parameter.	Write
x‘15’	MFC_Cmd MFC_ClassID	MFC Command Opcode Channel (see page 117). Write and enqueue MFC command with associated class ID. MFC Class ID Channel (see page 118). Write and enqueue MFC command with associated command opcode.	Write blocking
MFC Tag Status Channels			
x‘16’	MFC_WrTagMask	MFC Write Tag-Group Query Mask Channel (see page 129). Write tag mask.	Write
x‘17’	MFC_WrTagUpdate	MFC Write Tag Status Update Request Channel (see page 132). Write request for conditional or unconditional tag-status update.	Write blocking
x‘18’	MFC_RdTagStat	MFC Read Tag-Group Status Channel (see page 133). Read tag status (with mask applied).	Read blocking
x‘19’	MFC_RdListStallStat	MFC Read List Stall-and-Notify Tag Status Channel (see page 135). Read MFC list stall-and-notify status.	Read blocking
x‘1A’	MFC_WrListStallAck	MFC Write List Stall-and-Notify Tag Acknowledgment Channel (see page 136). Write MFC list stall-and-notify acknowledgment.	Write
x‘1B’	MFC_RdAtomicStat	MFC Read Atomic Command Status Channel (see page 137). Read atomic command status.	Read blocking
SPU Mailboxes			
x‘1C’	SPU_WrOutMbox	SPU Write Outbound Mailbox Channel (see page 139). Write outbound SPU mailbox contents.	Write blocking

Table B-1. SPU Channel Map (Page 3 of 3)

Channel Number (Hexadecimal)	Channel Name	Description	Access Type
x‘1D’	SPU_RdInMbox	SPU Read Inbound Mailbox Channel (see page 141). Read inbound SPU mailbox contents.	Read blocking
x‘1E’	SPU_WrOutIntrMbox	SPU Write Outbound Interrupt Mailbox Channel (see page 140). Write SPU outbound interrupt mailbox contents.	Write blocking
x‘1F’:x‘3F’	Channel 31—Channel 63	Reserved	

Appendix C. CBEA-Specific PPE Special Purpose Registers

Table C-1 lists the PowerPC Processor Element (PPE) special-purpose registers (SPRs) required by the Cell Broadband Engine Architecture (CBEA). These registers are accessed using the move to special-purpose register (**mtspr**) instruction and move from special-purpose register (**mfspr**) PowerPC instruction.

Note: This is not a complete list of SPRs. There can be additional SPRs for specific implementations. See *Book III* of the *PowerPC Architecture* or the specific implementation documentation for a complete list of SPRs.

Table C-1. PPE Special Purpose Register Map (Page 1 of 2)

SPR Number	Register	Description	Access Type
Version Register			
x'3FE'	BP_VR	CBEA-Compliant Processor Version Register (see page 287).	Read Only
Replacement-Management Table Area (RMT) (Implementation-dependent area: See the specific implementation documentation for more information)			
x'3B6'	PPE_TLB_RMT_Index	RMT Index Register (see page 257). Index of the RMTs.	Read/Write
x'3B7'	PPE_TLB_RMT_Data	RMT Data Register (see page 258). Doubleword of RMT data pointed to by the RMT Index Register. Entry contents are implementation dependent.	Read/Write
Instruction Range SPRs			
x'3D0'	IRSR0	Instruction Range-Start Register 0 (duplicated per thread).	Read/Write
x'3D1'	IRMR0	Instruction Range-Mask Register 0 (duplicated per thread).	Read/Write
x'3D2'	ICIDR0	Instruction Class ID Register 0 (duplicated per thread).	Read/Write
x'3D3'	IRSR1	Instruction Range-Start Register 1 (duplicated per thread).	Read/Write
x'3D4'	IRMR1	Instruction Range-Mask Register 1 (duplicated per thread).	Read/Write
x'3D5'	ICIDR1	Instruction Class ID Register 1 (duplicated per thread).	Read/Write
Data Range SPRs			
x'3B8'	DRSR0	Data Range-Start Register 0 (duplicated per thread).	Read/Write
x'3B9'	DRMR0	Data Range-Mask Register 0 (duplicated per thread).	Read/Write
x'3BA'	DCIDR0	Data Class ID Register 0 (duplicated per thread).	Read/Write
x'3BB'	DRSR1	Data Range-Start Register 1 (duplicated per thread).	Read/Write
x'3BC'	DRMR1	Data Range-Mask Register 1 (duplicated per thread).	Read/Write
x'3BD'	DCIDR1	Data Class ID Register 1 (duplicated per thread).	Read/Write

1. Reading of these registers should be allowed for diagnostic purposes.

Table C-1. PPE Special Purpose Register Map (Page 2 of 2)

SPR Number	Register	Description	Access Type
Translation Lookaside Buffer (TLB) Management SPRs			
x‘3B2’	PPE_TLB_Index_Hint	TLB Index Hint Register (see page 209). Index of best TLB entry to update.	Read Only
x‘3B3’	PPE_TLB_Index	TLB Index Register (see page 210). Index of TLB entry to update with TLB Real Page Number Register and TLB Virtual Page Number Register.[1]	Write Only
x‘3B4’	PPE_TLB_VPN	TLB Virtual Page Number Register (see page 211). Access to upper portion of TLB entry.	Read/Write
x‘3B5’	PPE_TLB_RPN	TLB Real Page Number Register (see page 212). Access to lower portion of TLB entry.	Read/Write

1. Reading of these registers should be allowed for diagnostic purposes.

Appendix D. Defined Commands

This appendix contains the tables of defined commands from *Section 7 MFC Commands* beginning on page 55. These are the same tables that are shown in that section.

Each of these commands can contain one or more of the parameters listed in *Table D-1 Parameter Mnemonics*.

Defined commands fall into one of three categories:

- Data transfer commands are shown in *Table D-2 Data Transfer or MFC DMA Commands* on page 312.

  The data transfer commands are further divided into subcategories which define the direction of the data movement (that is, to or from local storage, or **get** and **put**). An application can place the data transfer commands listed in *Table D-2* into the command queue. Unless otherwise noted, these commands can be executed out of order.
- SL1 cache-management commands are shown in *Table D-3 SL1 Storage Control Commands* on page 313.
- Synchronization commands are shown in *Table D-4 MFC Synchronization Commands* on page 314.

Note: Embedded fencing and synchronization commands *must* be used to ensure proper ordering when ordering is required.

Table D-1 lists the parameter mnemonics.

Table D-1. Parameter Mnemonics

Parameter	Parameter Name	Register Name	See Note
CL	MFC Class ID	MFC_ClassID	
TG	MFC Command Tag Identification	MFC_Tag	
TS	MFC Transfer Size	MFC_Size	1
LSZ	MFC List Size	MFC_Size	1
LSA	MFC Local Storage Address	MFC_LSA	
EAH	MFC Effective Address High	MFC_EAH	4
EAL	MFC Effective Address Low	MFC_EAL	2
LA	MFC List Local Storage Address	MFC_EAL	2
LTS	List Element Transfer Size		3
LEAL	List Element Effective Address Low		3

1. TS and LSZ share the same register offset. The meaning of the contents depends on the command modifier of the MFC opcode.
2. EAL and LA share the same register offset. The meaning of the contents depends on the command modifier of the MFC opcode.
3. No associated registers. These parameters are located in local storage and are referenced by the list address (LA) parameter.
4. This parameter is optional.

Table D-2 shows the data transfer, or direct memory access (DMA) commands available in the Cell Broadband Engine Architecture (CBEA).

Table D-2. Data Transfer or MFC DMA Commands (Page 1 of 2)

Mnemonic	Opcode	Support (Proxy/Channel)	Description
Put Commands			
put	x‘0020’	Proxy/Channel	Moves data from local storage to an effective address within the main storage domain.
puts	x‘0028’	Proxy	Moves data from local storage to an effective address within the main storage domain. Starts the SPU after the DMA operation completes.
putr	x‘0030’	Proxy/Channel	Same as **put** with a PPE L2 cache scarf hint (used to send results to a PPE).[1]
putf	x‘0022’	Proxy/Channel	Moves data from local storage to an effective address within the main storage domain with fence. This command is locally ordered with respect to all previously issued commands within the same tag group and command queue.
putb	x‘0021’	Proxy/Channel	Moves data from local storage to an effective address within the main storage domain 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	x‘002A’	Proxy	Moves data from local storage to an effective address within the main storage domain with fence. This command is locally ordered with respect to all previously issued commands within the same tag group and command queue. Starts the SPU after the DMA operation completes.
putbs	x‘0029’	Proxy	Moves data from local storage to an effective address within the main storage domain 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. Starts the SPU after the DMA operation completes.
putrf	x‘0032’	Proxy/Channel	Same as **putf** with a PPE L2 cache scarf hint used to send results to a PPE.[1]
putrb	x‘0031’	Proxy/Channel	Same as **putb** with a PPE L2 cache scarf hint used to send results to a PPE. [1]
putl	x‘0024’	Channel	Moves data from local storage to an effective address within the main storage domain using an MFC list.
putrl	x‘0034’	Channel	Same as **putl** with a PPE L2 cache scarf hint used to send results to a PPE.[1]
putlf	x‘0026’	Channel	Moves data from local storage to an effective address within the main storage domain 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	x‘0025’	Channel	Moves data from local storage to an effective address within the main storage domain 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.
putrlf	x‘0036’	Channel	Same as **putlf** with a PPE L2 cache scarf hint used to send results to a PPE.[1]
putrlb	x‘0035’	Channel	Same as **putlb** with a PPE L2 cache scarf hint used to send results to a PPE.[1]
Get Commands			
get	x‘0040’	Proxy/Channel	Moves data from an effective address within the main storage domain to local storage.
gets	x‘0048’	Proxy	Moves data from an effective address within the main storage domain to local storage. Starts the SPU after DMA operation completes.

1. Scarfing is the direct transfer of data to a PPE L2 cache.

Table D-2. Data Transfer or MFC DMA Commands (Page 2 of 2)

Mnemonic	Opcode	Support (Proxy/Channel)	Description
getf	x'0042'	Proxy/Channel	Moves data from an effective address within the main storage domain to local storage with fence. This command is locally ordered with respect to all previously issued commands within the same tag group and command queue.
getb	x'0041'	Proxy/Channel	Moves data from an effective address within the main storage domain to local storage 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	x'004A'	Proxy	Moves data from an effective address within the main storage domain to local storage with fence. This command is locally ordered with respect to all previously issued commands within the same tag group. Starts the SPU after DMA operation completes.
getbs	x'0049'	Proxy	Moves data from an effective address within the main storage domain to local storage 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. Starts the SPU after DMA operation completes.
getl	x'0044'	Channel	Moves data from an effective address within the main storage domain to local storage using an MFC list.
getlf	x'0046'	Channel	Moves data from an effective address within the main storage domain to local storage 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	x'0045'	Channel	Moves data from an effective address within the main storage domain to local storage 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.

1. Scarfing is the direct transfer of data to a PPE L2 cache.

Table D-3 lists the storage control commands available for the first level DMA transfer cache (SL1).

Table D-3. SL1 Storage Control Commands

Mnemonic	Opcode	Support	Description
sdcrt	x'0080'	Proxy/Channel	Brings a range of effective addresses into the SL1 (performance hint for DMA gets).[1]
sdcrtst	x'0081'	Proxy/Channel	Brings a range of effective addresses into the SL1 (performance hint for DMA puts).[1]
sdcrz	x'0089'	Proxy/Channel	Writes zeros to the contents of a range of effective addresses.
sdcrst	x'008D'	Proxy/Channel	Stores the modified contents of a range of effective addresses.
sdcrf	x'008F'	Proxy/Channel	Stores the modified contents of a range of effective addresses and invalidates the block.

1. These commands do not transfer data in implementations without an SL1.

Table D-4 lists the synchronization commands available in the CBEA.

Table D-4. MFC Synchronization Commands

Command	Opcode	Support	Description
sndsig	x‘00A0’	Proxy/Channel	Updates signal notification registers in an I/O device or another SPU. This command is actually a 4-byte DMA put that can go to any address.
sndsigf	x‘00A2’	Proxy/Channel	Updates signal notification registers in an I/O device or another SPU with fence. This command is actually a 4-byte DMA put that can go to any address.
sndsigb	x‘00A1’	Proxy/Channel	Updates signal notification registers in an I/O device or another SPU with barrier. This command is actually a 4-byte DMA **put** that can go to any address.
barrier	x‘00C0’	Proxy/Channel	Barrier type ordering. Ensures ordering of all preceding, nonimmediate DMA commands with respect to all commands following the **barrier** command within the same command queue. The **barrier** command has no effect on the immediate DMA commands: **getllar**, **putllc**, and **putlluc**.
mfceieio	x‘00C8’	Proxy/Channel	The **mfceieio** command orders the storage transactions caused by **get** and **put** commands. To ensure that the commands are correctly ordered, the commands must be in the same tag group as the **mfceieio** command, or a barrier command must be issued before the **mfceieio** command. The **mfceieio** command orders transactions as follows, assuming the MFC DMA commands are within the specified tag group: • Orders **get** or **put** commands with respect to other **get** or **put** commands that access storage defined as caching inhibited and guarded. • Orders **put** commands that access storage defined as write through required with respect to **put** or **get** commands that access storage defined as caching inhibited and guarded. • Orders **put** or **get** commands that access storage defined as caching inhibited and guarded with respect to **put** commands that access storage defined as write through required. • Orders **put** commands with respect to **put** commands that access storage that is defined as memory coherency required and is neither write through required nor caching inhibited.
mfcsync	x‘00CC’	Proxy/Channel	Controls the ordering of DMA **put** and **get** operations within the specified tag group with respect to other processing units and devices in the system.

Table D-5 lists the atomic commands available in the CBEA.

Table D-5. MFC Atomic Commands

Command	Opcode	Support	Description
getllar	x‘00D0’	Channel	Gets a lock line and creates a reservation (executes immediately).
putllc	x‘00B4’	Channel	Puts lock line conditional on a reservation (executes immediately).
putlluc	x‘00B0’	Channel	Puts lock line unconditional (executes immediately).
putqlluc	x‘00B8’	Channel	Puts lock line unconditional (queued form).

Appendix E. Extensions to the PowerPC Architecture

This section describes instructions and facilities that are extensions to the PowerPC Architecture. They are not described in the *PowerPC Architecture* document. See the following sections for more information:

- *Section E.1 Software Management of TLBs (optional)*
- *Section E.2 Mediated External Exception Extension (optional)* on page 316
- *Section E.3 Multiple Concurrent Large Pages (optional)* on page 318
- *Section E.4 Defined Behavior for Inaccessible SPRs* on page 319
- *Section E.5 Vector/SIMD Multimedia Extension (optional)* on page 319

E.1 Software Management of TLBs (optional)

Optionally, an implementation can support software management of the translation lookaside buffers (TLBs) for both the PowerPC Processor Elements (PPEs) and Synergistic Processor Elements (SPEs). The facilities for software management of the SPE and PPE TLBs are described in *Section 14.3 Translation Lookaside Buffer Management* on page 207.

In addition to the facilities described in *Section 14.3*, the following PPE changes to the existing instructions and facilities defined in the PowerPC Architecture can be implemented:

- Add the translation lookaside buffer (TLB) Load Control bit (LPCR[TL]) to the Logical Partitioning Control Register (LPCR). The values for the TLB Load Control bit (LPCR[53]) follow:

  0 TLB is loaded by hardware.
  1 TLB is loaded by software.

- Add the invalidate selector (IS) field to the RB register of the **tlbiel** instruction. The values for the IS field (RB[52:53]) follow:

  00 The TLB is as selective as possible in invalidating the TLB entry. The implementation should use as many virtual page number (VPN) bits as possible, including the L, LP, and LPID, to eliminate invalidating unnecessary entries.

  01 The TLB entry is not invalidated. Any lower level caches of the translation are invalidated.

  10 The TLB does a congruency-class invalidate if the LPID matches the current value in the Logical Partition ID Register (LPIDR).

  11 The TLB does a congruency-class invalidate regardless of LPID match.

  Implementation Note: An implementation can choose to implement a subset of these options. It is always acceptable for an implementation to invalidate more TLB entries than specified by this instruction. The IS bits only provide a useful hint for a performance benefit.

E.2 Mediated External Exception Extension (optional)

Optionally, a PPE can support the mediated external exception extension to the PowerPC Architecture. On a shared processor (that is, a processor on which virtual partitions are dispatched), the mediated external exception extension can reduce external interrupt latency.

In the current PowerPC Architecture, external interrupts are disabled if the External Interrupt Enable bit in the Machine State Register (MSR[EE]) is set to '0'. Since MSR[EE] can be altered by an operating system, the presentation of an external interrupt for other partitions can be delayed by an arbitrary amount of time. This can affect the interrupt latency of other partitions. The mediated external exception extension addresses the interrupt latency issue by allowing the presentation of an external interrupt even if interrupts are disabled (MSR[EE] is set to '0').

The mediated external exception extension defines a new external exception called a "mediated external exception." The currently defined external exception is called a "direct external exception." An external interrupt that is caused by a mediated external exception is called a "mediated external interrupt." Correspondingly, an external interrupt that is caused by a direct external exception is called a "direct external interrupt."

The mediated external exception extension requires the following changes to the existing facilities described in the PowerPC Architecture:

- Define bit 52 of the Logical Partitioning Control Register (LPCR) as the Mediated External Exception Request (MER) bit. The values of the LPCR[MER] bit follow:

0	Mediated external exception is not requested.
1	Mediated external exception is requested.

- If the Logical Partitioning Environment Selector bit (LPCR[LPES0]) is set to '0' when a direct or mediated external interrupt occurs, save the state in Hypervisor Machine Status Save/Restore Register 0 (HSRR0) and Hypervisor Machine Status Save/Restore Register 1 (HSRR1). If LPCR[LPES0] is set to '1', save the state in Machine Status Save/Restore Register 0 (SRR0) and Machine Status Save/Restore Register 1 (SRR1).
- If LPCR[LPES0] is set to '0' when an external interrupt occurs, set HSRR1[42] to '1' for a mediated external interrupt; otherwise, set it to '0'. If LPCR[LPES0] is set to '1' when an external interrupt occurs, set SRR1[42] to '0'. 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.
- Define an implementation-specific enable bit for the mediated external exception extension. If this bit is not set, external interrupts are handled as defined in the PowerPC Architecture.
- Ensure that mediated external interrupts do not occur when LPCR[LPES0] is set to '1'. A hypervisor can accomplish this by setting LPCR[MER] to '0' whenever it sets LPCR[LPES0] to '1'. If a hypervisor violates this requirement, the results are undefined.

A signal from the internal interrupt controller (IIC) causes direct external exceptions (see *Section 21.2 Interrupt Presentation* on page 262). Mediated external exceptions are caused by a hypervisor setting the Mediated External Exception Request bit in the LPCR to '1' (LPCR[MER] = '1'). The equations for enabling both types of external exceptions follow.

- Direct external interrupts are enabled if the value of the following expression is ‘1’:

  ```
  MSR[EE] | (^(LPCR[LPES0]) & (^(MSR[HV]) | MSR[PR]))
  ```

  In particular, if LPCR[LPES0] = ‘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 following simpler expression is equivalent to the preceding expression:

  ```
  MSR[EE] | ^(LPCR[LPES0] | MSR[HV])
  ```

- Mediated external interrupts are enabled if the value of the following expression is ‘1’:

  ```
  MSR[EE] & (^(MSR[HV]) | MSR[PR])
  ```

  In particular, mediated external interrupts are disabled if the processor is in hypervisor state.

E.2.1 Using the Mediated External Exception Extension

This section provides an example of how the mediated external exception extension can be used in a shared processor with LPCR[LPES0] set to ‘0’. The example shows the conditions and steps necessary to redispatch a logical partition. In this example, an external interrupt has occurred for the currently executing virtual partition and has not yet been presented to the virtual partition.

Note: If the external interrupt has occurred for a *nonactive* logical partition and has not yet been presented to the partition, the hypervisor can save the currently executing partition’s state, restore the state of the partition for which the interrupt is intended, making it the currently executing virtual partition. This allows the interrupt to be handled with the minimal latency.

The example assumes that the external Real Mode Offset Register (RMOR) and the Hypervisor Real Mode Offset Register (HRMOR) are set to different values (see the implementation-specific documentation for more information about these registers). The operating system’s external interrupt handler is located at RMOR | x‘500’; the hypervisor’s external interrupt handler is located at HRMOR | x‘500’.

The hypervisor passes control to the operating system’s external interrupt handler as described below.

- If the external interrupt was direct (HSRR1[42] = ‘0’) and external interrupts are enabled for the partition (MSR[EE] = ‘1’), the hypervisor performs these steps:
  1. Sets LPCR[MER] to ‘0’.
  2. Sets registers (MSR, SRR0 and SRR1, and the external interrupt hardware registers as appropriate) to emulate the external interrupt.
  3. Returns to the operating system’s external interrupt handler (by executing the **hrfid** instruction with HSRR0 set to x‘500’ and HSRR1 set to the proper MSR value as required by the architecture).
- If the external interrupt was direct (HSRR1[42] = ‘0’) but external interrupts are disabled for the partition (MSR[EE] = ‘0’), the hypervisor performs these steps:
  1. Sets LPCR[MER] to ‘1’.
  2. Resets the PPE interrupting condition by performing a destructive read of the Interrupt Pending Port Register (see page 263), queueing the interrupt status for later presentation to the owning partition.
  3. Sets the Interrupt Current Priority Level Register (see page 268) to the least favored current priority of all virtual partitions.

4. Returns to the partition at the instruction at which it was interrupted (by executing the **hrfid** instruction).

   Note: In this section, it is assumed that LPCR[LPES0] is set to ‘0’. Therefore, it is necessary to return to the partition because the hypervisor is handling the external interrupt.

- If the external interrupt was mediated (HSRR1[42] = ‘1’) and external interrupts are now enabled for the partition (MSR[EE] = ‘1’), the hypervisor performs these steps:
  1. Sets registers (MSR, SRR0 and SRR1, and external interrupt hardware registers as appropriate) to emulate the original direct external interrupt.
  2. Returns to the operating system’s external interrupt handler (by executing the **hrfid** instruction with HSRR0 set to x‘500’ and HSRR1 set to the correct MSR value as required by the architecture).
  3. When the operating system’s external interrupt handler calls the hypervisor to obtain the interrupting condition and status, sets `LPCR[MER]` to ‘0’ if all external interrupts (for the partition) now have been presented to the partition. Otherwise, does not modify the `LPCR[MER]`. Returns from the hypervisor call with the dequeued interrupting condition and status.

In all three cases, the partition is redispatched with MSR[EE] set to ‘0’. Before the partition is redispatched to the operating system’s external interrupt handler (as in the first and third cases), the hypervisor sets the MSR and SRR0 and SRR1 as if the original direct external interrupt occurred when LPCR[LPES0] 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) about whether the external interrupt that is now being presented to the partition was direct, as in the first case, or mediated, as in the third case.

E.3 Multiple Concurrent Large Pages (optional)

Optionally, an implementation can support multiple concurrent large pages for the address translation mechanism. In the current PowerPC Architecture, only the default page size of 4 KB and one larger page size is supported. The CBEA extends the address translation mechanism of the PowerPC Architecture with support for up to eight concurrent large pages. The number of large pages that are supported and the size of the large pages is implementation dependent. Thus, an implementation can choose to only support 4 KB pages.

The following changes to the existing address translation facility described in the PowerPC Architecture can be implemented:

- Define reserved bits 58 and 59 of the segment lookaside buffer (SLB) as the LP field (SLB[LP]).
- The page size selected by the SLB is now the L bit concatenated with the LP field (SLB[L] || SLB[LP]).
- The real page number (RPN) field of the page table entry (PTE) is now split into two fields: abbreviated real page number (ARPN) and LP.
- The L bit in the SLB no longer has to match the L bit in the TLB.

The new LP field in the SLB changes the definition of the **slbmte** and **slbmfev** PowerPC instructions. Bits 58 and 59 of the RS register are now defined as the LP field for the **slbmte** instruction. Bits 58 and 59 of the RT register are now defined as the LP field for the **slbmfev** instruction.

The lower 8 bits of the RPN in the page table entry are now defined as the large page selector (LP). The remaining upper bits are now defined as the ARPN. When the L bit in the PTE is ‘0’, the page size is 4 KB. The RPN is the ARPN field concatenated with the LP field (ARPN || LP). When the L bit is ‘1’, the lower bits of the LP field select the page size; the ARPN concatenated with the remaining upper bits of the LP field is the real page number (RPN). The number of upper bits used from the LP field is dependent on the page size selected. The definition of the LP field in the PTE is the same as the LP field in the TLB Real Page Number Register (see page 212).

When searching the page table, the L bit in the SLB no longer has to match the L bit in the TLB. Instead, the page size selected by the SLB (SLB[L] || SLB[LP]) must match the page size selected by TLB[L] || TLB[LP] for the PTE to match the effective address.

E.4 Defined Behavior for Inaccessible SPRs

The current PowerPC Architecture allows an implementation boundedly undefined behavior when an inaccessible special purpose register (SPR) is accessed using the **mtspr** or **mfspr** instructions. The CBEA further defines the behavior when inaccessible SPRs are accessed. Specifically, when the processor is in problem state, either a privileged or illegal instruction interrupt occurs, depending on the value of the high-order bit of the SPR number. When the processor is in privileged state, the instruction is executed as a no-op. Executing the instruction as a no-op when the processor is in privileged state allows an operating system to unconditionally save and restore nonprivileged SPRs that may not be implemented on some processors, regardless of the processor on which it is executing.

When reading unimplemented or write only SPRs, an implementation is allowed to return zeros instead of treating the read as a no-op. If an implementation returns zeros, it is not required to cause an interrupt when the processor is in problem state and the higher order bit of the SPR number is zero. However, implementation of the behavior described in the preceding paragraph is strongly recommended.

E.5 Vector/SIMD Multimedia Extension (optional)

To improve the performance of a PPE for multimedia applications, the PPEs can support the vector/single instruction, multiple data (SIMD) multimedia extension to the PowerPC Architecture. The vector/SIMD multimedia extension enhances the current PowerPC instruction set with 4-way SIMD instructions. For a definition of the instructions, see the *PowerPC Microprocessor Family: Vector/SIMD Multimedia Extension Technology Programming Environments Manual*, version 2.07c.

Appendix F. Examples of Access Ordering

This appendix contains examples of access ordering as discussed in *Section 10* beginning on page 175.

The following examples demonstrate the order of accesses originated by a synergistic processor unit (SPU), a memory flow controller (MFC), or a PowerPC Processor Element (PPE). In these examples, MFC, SPU, and PPE are each shown executing a sequence of instructions or commands. No order between sequences is implied merely by the relative line position of any instruction or command in one sequence compared to the line position of an instruction or command in another sequence. Unless otherwise stated, main storage locations in these examples have main storage attributes of coherence required and not caching inhibited. Local storage alias in these examples have main storage attributes of caching inhibited when accessed from the main storage domain.

See *Section 8.8.1 MFC Multisource Synchronization Register* beginning on page 109 for additional examples of cumulative ordering across the local storage and main storage domains.

Example 1: putf to main storage

SPU (0)

1. Store to local storage location A.
2. **dsync**
3. Store to local storage location B.

MFC (0)

1. **put** TG = ‘1’: copy local storage location B to the main storage location D.
2. **putf** TG = ‘1’: copy local storage location A to the main storage location C.

PPE (0)

1. Load main storage location C.
2. Use any serialization mechanism that causes these loads to be performed in order (see *PowerPC Architecture, Books I-III*).
3. Load main storage location D.

If main storage location D is written with the new value of local storage location B, then main storage location C is written with the new value of local storage location A. After PPE (0) loads the new value of location C, it must load the new value of location D.

Example 2: putf to remote local storage

SPU (0)

1. Store to local storage location A.
2. **dsync**
3. Store to local storage location B.

MFC (0)

1. **put** TG = ‘1’: copy local storage location B of SPU (0) to local storage location D of SPU (1).
2. **putf** TG = ‘1’: copy local storage location A of SPU (0) to local storage location C of SPU (1).

SPU (1)

1. Load local storage location C.
2. **dsync**
3. Load local storage location D.

If local storage location D is written with the new value of local storage location B, then local storage location C is written with the new value of local storage location A. When SPU (1) loads the new value of local storage location C, it must load the value of local storage location D.

Example 3: getf versus PPE stores

MFC (0)

1. **get** TG = ‘1’: read main storage location A.
2. **getf** TG = ‘1’: read main storage location B.

PPE (0)

1. Store to main storage location B.
2. Use any serialization mechanism that causes these stores to be performed in order (see *PowerPC Architecture, Books I-III*).
3. Store to main storage location A.

If MFC (0) gets the new value of location A, it must get the new value of B.

Example 4: getf versus remote SPU stores

MFC (0)

1. **get** TG = ‘1’: read local storage location A of SPU (1).
2. **getf** TG = ‘1’: read local storage location B of SPU (1).

SPU (1)

1. Store to local storage location B.
2. **dsync**
3. Store to local storage location A.

If MFC (0) gets the new value of location A, it must get the new value of B.

Example 5: getf versus local SPU loads

This example only demonstrates storage access ordering. It is not a recommended programming method. Typically, software would use the tag group completion facility to get predictable new values for the SPU loads.

MFC (0)

1. **get** TG = ‘1’: copy main storage location C to local storage location A of SPU (0).
2. **getf** TG = ‘1’: copy main storage location D to local storage location B of SPU (0).

SPU (0)

1. Load from local storage location B.
2. **dsync**
3. Load from local storage location A.

If SPU (0) loads the new value of location B, it must also load the new value of A.

Example 6: Tag-independent **barrier** command versus local SPU loads

MFC (0)

1. **get** TG = ‘1’: copy main storage location X to local storage location A.
2. **barrier** TG = ‘2’.
3. **get** TG = ‘3’: copy main storage location Y to local storage location B.

SPU (0)

1. Load from local storage location B.
2. **dsync**
3. Load from local storage location A.

If SPU (0) loads the new value of location B, it must load the new value of A.

Example 7: Tag-group status completion and **put** versus PPE loads

This sequence assumes the MFC Write Tag-Group Query Mask Channel (see page 129) is already set so that tag group 1 is part of a query.

SPU (0)

1. Enqueue a **put** TG = ‘1’ to write main storage location A.
2. Request an immediate tag-status update and wait for tag group completion.
   For more information, see *Section 9.3 MFC Tag-Group Status Channels* on page 126, and *Section 9.3.5 MFC Write Tag Status Update Request Channel* on page 132.
3. Enqueue a **put** TG = ‘1’ to write main storage location B.

PPE (0)

1. Load from main storage location B.
2. Use any serialization mechanism that orders these loads.
3. Load from main storage location A.

If PPE (0) loads the new value of location B, PPE(0) can load either the old or the new value of A. The DMA accesses are ordered with respect to local storage of SPU (0) by the MFC Read Tag-Group Status completion status but are not ordered with respect to the main storage domain.

When the status returned by a channel read instruction that targets the MFC Read Tag-Group Status Channel (see page 133) indicates that a queued **put** or **get** command is complete, the MFC local storage address (LSA) access is complete. See *Section 10 Storage Access Ordering* on page 175 for more information.

Example 8: Tag-group status completion and **put** versus local SPU loads

This sequence assumes the MFC Write Tag-Group Query Mask Channel (see page 129) is already set so that tag group 1 is part of a query.

SPU (0)

1. Enqueue a **put** TG = ‘1’ to write main storage location A.
2. Request an immediate tag-status update and wait for tag group completion.
   For more information, see *Section 9.3 MFC Tag-Group Status Channels* on page 126, and *Section 9.3.5 MFC Write Tag Status Update Request Channel* on page 132.
3. Enqueue a **put** TG = ‘1’ to write main storage location B.

SPU (1)

1. Load from local storage location B.
2. **dsync**
3. Load from local storage location A.

If SPU (1) loads the new value of location B, it can load either the old or the new value of A. The DMA accesses for each **put** command are locally ordered with respect to local storage of SPU (0) by the tag-group completion status, but they are not ordered with respect to the local storage of another SPU.

When the status returned by a channel read instruction that targets the MFC Read Tag-Group Status Channel (see page 133) indicates that a queued **put** or **get** command is complete, the MFC local storage address (LSA) access is complete. See *Section 10 Storage Access Ordering* on page 175 for more information.

Example 9: mfcsync and **put** versus PPE loads

MFC (0)

1. **put** TG = '1' to write main storage location A.
2. **mfcsync** TG = '1'
3. **put** TG = '1' to write main storage location B.

PPE (0)

1. Load main storage location B.
2. Use any serialization mechanism that causes these loads to be performed in order (see *PowerPC Architecture, Books I-III.*)
3. Load main storage location A.

If PPE (0) loads the new value of location B, it must also load the new value of A.

Example 10: mfcsync and **put** versus remote SPU local storage loads

MFC (0)

1. **put** TG = '1' to write local storage location A of SPU (1).
2. **mfcsync** TG = '1'
3. **put** TG = '1' to write local storage location B of SPU (1).

SPU (1)

1. Load from local storage location B.
2. **dsync**
3. Load from local storage location A.

If SPU (1) loads the new value of location B, it must also load the new value of A.

Example 11: mfcsync and **put** versus local SPU local storage stores

This example only demonstrates storage access ordering. If it is only necessary to ensure local storage access order and not main storage access order, then a tag-specific fence or barrier is sufficient.

MFC (0)

1. **put** TG = '1' to read from local storage location A.
2. **mfcsync** TG = '1'
3. **put** TG = '1' to read from local storage location B.

SPU (0)

1. Store to local storage location B.
2. **dsync**
3. Store to local storage location A.

If MFC (0) reads the new value of location A, it must also read the new value of B.

Example 12: mfcsync and **get**

MFC (0)

1. **get** TG = ‘1’ to read from main storage location A.
2. **mfcsync** TG = ‘1’
3. **get** TG = ‘1’ to read from main storage location B.

PPE (0)

1. Store to main storage location B.
2. Use any serialization mechanism that causes these stores to be performed in order (see *PowerPC Architecture, Books I-III*).
3. Store to main storage location A.

If MFC (0) gets the new value of location A, it must also get the new value of B.

Example 13: Cumulative Ordering

MFC (0)

1. **put** to write main storage location A.

PPE (0)

1. Load from main storage location A.
2. **sync**
3. Store to main storage location B.

MFC (1)

1. **get** TG = ‘1’ to read main storage location B.
2. **mfcsync** TG = ‘1’
3. **get** TG = ‘1’ to read main storage location A.

If PPE (0) loads the new value of location A and MFC (1) gets the new value of B, MFC (1) must also get the new value of A.

Example 14: Cumulative Ordering

MFC (0)

1. **put** TG = ‘1’ to write main storage location A.
2. **mfcsync** TG = ‘1’
3. **put** TG = ‘1’ to write main storage location B.

PPE (0)

1. Loop loading from main storage location B until the new value is loaded.
2. Store to main storage location C.

MFC (1)

1. **get** TG = ‘1’ to read main storage location C.
2. **mfcsync** TG = ‘1’
3. **get** TG = ‘1’ to read main storage location A.

If MFC (1) gets the new value of C, MFC (1) must also get the new value of A.

Example 15: SPU stores versus PPE loads

SPU (1)

1. Store to local storage location A.
2. **dsync**
3. Store to local storage location B.

PPE (0)

1. Load local storage location B.
2. Use any serialization mechanism that causes these loads to be performed in order (see *PowerPC Architecture, Books I-III.*)
3. Load local storage location A.

If PPE (0) loads the new value of location B, it must also load the new value of location A.

Example 16: SPU loads versus PPE stores

SPU (1)

1. Load from local storage location B.
2. **dsync**
3. Load from local storage location A.

PPE (0)

1. Store to local storage location A.
2. Use any serialization mechanism that causes these stores to be performed in order (see *PowerPC Architecture, Books I-III*).
3. Store to local storage location B.

If SPU (1) loads the new value of location B, it must also load the new value of A.

Example 17: SPU loads versus PPE stores to local storage and SPU Signal Notification 1 Register.

In this example, if more than two units are involved in the set of storage accesses that must be completed before the store to the SPU Signal Notification 1 Register, then MFC Multisource Synchronization Register (see page 109) must be used.

SPU (1)

1. Read from SPU Signal Notification 1 Register.
2. **dsync**
3. Load from local storage location A.

PPE (0)

1. Store to local storage location A.
2. Use any serialization mechanism that causes these stores to be performed in order (see *PowerPC Architecture, Books I-III*).
3. Store to SPU Signal Notification 1 Register.

If SPU (1) loads the new value of the SPU Signal Notification 1 Register, it must also load the new value of A.

Example 18: SPU loads versus PPE stores to local storage and SPU Inbound Mailbox Register (see page 103)

SPU (1)

1. Read from SPU Inbound Mailbox Register.
2. **dsync**
3. Load from local storage location A.

PPE (0)

1. Store to local storage location A.
2. Use any serialization mechanism that causes these stores to be performed in order (see *PowerPC Architecture, Books I-III*).
3. Store to SPU Inbound Mailbox Register.

If SPU (1) loads the new value of the SPU Inbound Mailbox Register, it must also load the new value of A.

Example 19: mfceieio and **put** versus PPE loads

MFC (0)

1. **put** TG = ‘1’ to write main storage location A.
2. **mfceieio** TG = ‘1’
3. **put** TG = ‘1’ to write main storage location B.

PPE (0)

1. Load main storage location B.
2. Use any serialization mechanism that causes these loads to be performed in order (see *PowerPC Architecture, Books I-III*.)
3. Load main storage location A.

If PPE (0) loads the new value of location B, it must also load the new value of A.

Example 20: mfceieio and **get**

Main storage locations A and B have the storage attributes of caching inhibited and guarded.

MFC (0)

1. **get** TG = ‘1’ to read from main storage location A.
2. **mfceieio** TG = ‘1’
3. **get** TG = ‘1’ to read from main storage location B.

PPE (0)

1. Store to main storage location B.
2. **eieio**
3. Store to main storage location A.

If MFC (0) gets the new value of location A, it must also get the new value of B.

Glossary

acknowledgment	A transmission that is sent as an affirmative response to a data transmission.
architecture	A detailed specification of requirements for a processor or computer system. It does not specify details of how the processor or computer system must be implemented; instead it provides a template for a family of compatible implementations.
barrier	1. A global command that ensures ordering of all preceding, nonimmediate DMA commands with respect to all commands following the **barrier** command within the same command queue. 2. A tag-specific MFC command modifier that ensures that the command with the barrier modifier and all subsequent commands with the same tag ID are locally ordered with respect to all previously issued commands within the same tag group and command queue. Compare to *fence*.
BIC	Bus interface controller. Part of the Cell Broadband Engine interface (BEI) to I/O.
big endian	A byte-ordering method in memory where the address n of a word corresponds to the most-significant byte. In an addressed memory word, the bytes are ordered (left to right) 0, 1, 2, 3, with 0 being the most-significant byte. See little endian.
bisled	branch indirect and set link if external data instruction
BIU	Bus interface unit. Part of the PPE interface to the EIB.
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.
caching inhibited	A memory update policy in which the cache is bypassed, and the load or store is performed to or from system memory. A page of storage is considered caching inhibited when the “I” bit has a value of ‘1’ in the page table. Data located in caching inhibited pages cannot be cached at any memory hierarchy that is not visible to all processors and devices in the system. Stores must update the memory hierarchy to a level that is visible to all processors and devices in the system.
CBEA	See Cell Broadband Engine Architecture.
Cell Broadband Engine Architecture	Extends the PowerPC 64-bit architecture with loosely coupled cooperative off-load processors. The Cell Broadband Engine Architecture provides a basis for the development of microprocessors targeted at the game, multimedia, and real-time market segments. The Cell Broadband Engine is one implementation of the Cell Broadband Engine Architecture.

channel	Channels are unidirectional, function-specific registers or queues. They are the primary means of communication between an SPE's SPU and its MFC, which in turn mediates communication with PPEs, other SPEs, and other devices. These other devices use MMIO registers in the destination SPE to transfer information on the channel interface of that destination SPE. Specific channels have read or write properties, and blocking or nonblocking properties. Software on the SPU uses channel commands to enqueue DMA commands, query DMA and processor status, perform MFC synchronization, access auxiliary resources such as the decrementer (timer), and perform interprocessor-communication via mailboxes and signal-notification.
CL	The class ID parameter in an MFC command.
coherence	Refers to memory and cache 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.
CSA	context-save area
CSRA	context save and restore area
DAR	Data Address Register
data storage interrupt	An interrupt posted when a fault is encountered accessing storage or I/O space. A typical data storage interrupt is a page fault or protection violation.
dcbf	data cache block flush instruction
dcbst	data cache block store instruction
dcbt	data-cache block touch x-form instruction
dcbtst	data cache block touch for store instruction
dcbz	data cache block zero instruction
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.
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.
DMA command	A type of MFC command that transfers or controls the transfer of a memory location containing data or instructions.

DMA queue	A storage area in which DMA commands are held until they complete. There are two types of queues: the MFC proxy command queue and the SPU command queue. The MFC proxy command queue holds commands issued by another processor or device. The SPU command queue holds commands issued by the corresponding SPU.
DMAC	Direct memory access controller. A controller that performs DMA transfers.
DR	data relocate
DSI	See data storage interrupt.
DSISR	Data Storage Interrupt Status Register
EAH	MFC effective address high
EAL	MFC effective address low
ECC	See error correction code.
effective address	An address generated or used by a program to reference memory. A memory-management unit translates an effective address to a virtual address, 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 PPEs, SPEs, memory, and I/O devices (or another Cell Broadband Engine).
EIC	external interrupt controller
eieio	enforce in-order execution of I/O transaction instruction
ERAT	Effective-to-real-address translation, or a buffer or table that contains such translations, or a table entry that contains such a translation.
error correction code	A code appended to a data block that can detect and correct bit errors within the block.
ESID	effective segment ID
exception	An error, unusual condition, or external signal that can alter a status bit and will cause a corresponding interrupt, if the interrupt is enabled. See interrupt.
fence	A tag-specific MFC command modifier that causes the MFC to wait for completion of all previously issued MFC commands within the same tag group and the same command queue before starting the MFC command with the fence option. It does not apply to subsequently issued commands or the immediate commands: **getllar**, **putllc**, and **putlluc**. Compare to *barrier*.
fetch	Retrieving instructions from either the cache or system memory and placing them into the instruction queue.
FIFO	First in, first out. Refers to one way elements in a queue are processed. It is analogous to “people standing in line.”

FLIH	first-level interrupt handler
floating point	A way of representing real numbers (that is, values with fractions or decimals) in 32 bits or 64 bits. Floating-point representation is useful to describe very small or very large numbers.
fres	floating reciprocal estimate single A-form instruction
frsqte	floating reciprocal square-root estimate A-form instruction
general purpose register	An explicitly addressable register that can be used for a variety of purposes (for example, as an accumulator or an index register).
getllar	get lock line and reserve command
GPR	See general purpose register.
guarded	Prevented from responding to speculative loads and instruction fetches. The operating system typically implements guarding, for example, on all I/O devices.
HID	hardware implementation dependent
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 Cell Broadband Engine 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. A PPE can then operate a subset of the SPEs in the Cell Broadband Engine with the real-time operating system, while the other SPEs run under the traditional operating system.
IABR	Instruction Address Breakpoint Register
icbi	instruction cache block invalidate instruction
IDEA	implementation-dependent expansion area
IGP	interrupt generation port
IIC	internal interrupt controller
implementation	A particular processor that conforms to the architecture but may differ from other architecture-compliant implementations for example in design, feature set, and implementation of optional features.
instruction cache	A cache for providing program instructions to the processor faster than they can be obtained from system memory.
INT	See interrupt.
interrupt	A change in machine state in response to an exception. See exception.

interrupt packet	Used to signal an interrupt, typically to a processor or to another interruptible device.
IOC	I/O controller
IOIF	Cell Broadband Engine I/O Interface. The EIB’s noncoherent protocol for interconnection to I/O devices.
IPI	interprocessor interrupt
IR	instruction relocate
ISA	instruction set architecture
ISRC	interrupt source
JTAG	Joint Test Action Group
KB	kilobyte
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.
LA	A local storage (LS) address of an MFC list. It is used as a parameter in an MFC command.
ld	load doubleword instruction
ldarx	load doubleword and reserve x-form instruction
LEAL	list element effective address low
least recently used	A policy for a caching algorithm that removes from the cache the item that has the longest elapsed time since its last access. An algorithm used to identify and make available the cache space that contains the data that was least recently used.
least-significant bit	The bit of least value in an address, register, data element, or instruction encoding.
little endian	A byte-ordering method in memory where the address *n* of a word corresponds to the least-significant byte. In an addressed memory word, the bytes are ordered (left to right) 3, 2, 1, 0, with 3 being the most-significant byte. See big endian.
livelock	An endless loop in program execution.
lmw	load multiple word instruction
local storage	The storage associated with each SPE. It holds both instructions and data.
logical partitioning	A function of an operating system that enables the creation of logical partitions.
LPAR	See logical partitioning.
LPID	logical-partition identity

LRU	See least recently used.
LS	See local storage.
LSA	Local storage address. An address in the LS of an SPU, by which programs running in the SPU and DMA transfers managed by the MFC access the LS.
LSb	See least-significant bit.
LSCA	local storage compare address
LSCAM	local storage compare address mask
LSCSA	local storage context save area
lswi	load string word immediate instruction
lswx	load string word indexed instruction
LSZ	MFC list size
LTS	list element transfer size
lwarx	load word and reserve x-form instruction
mailbox	A queue in an SPE's MFC for exchanging 32-bit messages between SPEs, PPEs, or other devices. Two mailboxes (the SPU Write Outbound Mailbox and SPU Write Outbound Interrupt Mailbox) are provided for sending messages from an SPE. One mailbox (the SPU Read Inbound Mailbox) is provided for sending messages to an SPE.
main storage	The effective-address space. It consists physically of real memory (whatever is external to the memory-interface controller), SPU LSs, memory-mapped registers and arrays, memory-mapped I/O devices, and pages of virtual memory that reside on disk. It does not include caches or execution-unit register files. See local storage.
mask	A pattern of bits used to accept or reject bit patterns in another set of data. Hardware interrupts are enabled and disabled by setting or clearing a string of bits, with each interrupt assigned a bit position in a mask register
MB	megabyte
memory coherency	An aspect of caching in which it is ensured that an accurate view of memory is provided to all devices that share system memory.
memory mapped	Mapped into the Cell Broadband Engine's addressable-memory space. Registers, SPE local storage (LS), I/O devices, and other readable or writable storage can be memory-mapped. Privileged software does the mapping.
MFC	Memory flow controller. It is part of an SPE and provides two main functions: moves data via DMA between SPE local storage (LS) and main storage, and synchronizes the SPU with the rest of the processing units in the system.

mfceieio	MFC enforce in-order execution of I/O command
mfcsync	MFC synchronize command
mfspr	move from special-purpose register instruction
MIC	Memory interface controller. The Cell Broadband Engine's MIC supports two memory channels.
MMIO	Memory-mapped input/output. 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.
most recently used	A policy for a caching algorithm that removes from the cache the item that has the shortest elapsed time since its last access., An algorithm used to identify and make available the cache space that contains the data that was most recently used.
most-significant bit	The highest-order bit in an address, registers, data element, or instruction encoding.
MRU	See most recently used.
MSb	most-significant bit
MSR	Machine State Register
mtmsr	move to machine state register instruction
mtspr	move to special-purpose register instruction
no-op	No-operation. A single-cycle operation that does not affect registers or generate bus activity.
page	A region in memory. The PowerPC Architecture defines a page as a 4 KB area of memory, aligned on a 4 KB boundary or a large page size which is implementation dependent.
page table	A table that maps virtual addresses (VAs) to real addresses (RAs) and contains related protection parameters and other information about memory locations.
PMD	power management and debug area
PME	privileged mode environment
POR	power-on reset
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 reduced instruction set computer (RISC) processors. The PowerPC Architecture was developed jointly by Apple, Motorola, and IBM.

PPE	PowerPC Processor Element. A general-purpose processor in the Cell Broadband Engine. Consists of the PPU and the PPSS.
PPU	PowerPC processor unit. The part of a PPE that includes execution units, memory-management unit, and the L1 cache.
privileged mode	Also known as supervisor mode. The permission level of operating system instructions. The instructions are described in *PowerPC Architecture, Book III* and are required of software that accesses system-critical resources.
privileged software	Software that has access to the privileged modes of the architecture.
problem state	The permission level of user instructions. The instructions are described in *PowerPC Architecture, Books I and II* and are required of software that implements application programs.
PTE	Page table entry. See page table.
putllc	put lock line conditional command
putlluc	put lock line unconditional command
putqlluc	put queued lock line unconditional command
PVR	Processor Version Register
QoS	Quality of service. This usually relates to a guarantee of minimum bandwidth for streaming applications.
quadword	A group of 16 contiguous locations starting at an address divisible by 16.
RA	real address
RAG	resource allocation group
RAID	resource allocation ID
RAM	resource allocation management
rchcnt	read channel count instruction
RClassID	replacement class identifier
rdch	read from channel instruction
real address	An address for physical storage, which includes physical memory, a PPE's L1 and L2 caches, and SPE local storage (LS) if the operating system has mapped the LSs to the real-address space. The maximum size of the real-address space is 2^{42} bytes.
RFC	request for change
RMT	replacement management table
RNG	random number generation

RPN	real page number
scarfing	The direct transfer of data to the PPE L2 cache
SCEI	Sony Computer Entertainment Incorporated
sdcrf	SL1 data cache range flush instruction
sdcrst	SL1 data cache range store instruction
sdcrt	SL1 data cache range touch instruction
sdcrtst	SL1 data cache range touch for store instruction
sdcrz	SL1 data cache range set to zero instruction
SDR	Storage Descriptor Register
SG	SPU group
signal	Information sent on a signal-notification channel. These channels are inbound (to an SPE) registers. They can be used by a 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.
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.
SL1	A first-level cache for DMA transfers between local storage and main storage
SLB	Segment lookaside buffer. It is used to map an effective address to a virtual address.
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
SLI	state lost and isolated
sndsig	send signal command
sndsigb	update signal-notification registers in an I/O device or another SPU with barrier command

sndsigf	update signal-notification registers in an I/O device or another SPU with fence command
snoop	To compare an address on a bus with a tag in a cache to detect operations that violate memory coherency.
SPE	Synergistic Processor Element. Consists of a synergistic processor unit (SPU), a memory flow controller (MFC), and local storage (LS).
SPR	special purpose register
SPU	Synergistic processor unit. The part of an SPE that executes instructions from its local storage (LS).
SRI	state retained and isolated
SRR0/SRR1	Save and Restore Register 0 and 1
stdcx	store doubleword conditional indexed instruction
stmw	store multiple word instruction
storage model	A CBEA-compliant processor implements two concurrent storage models for an application program: the virtual storage model of the PPE (also used by MFCs for DMA operations), and the local storage model of the SPU. For more information about storage models, see *Section 3* on page 41.
stswi	store string word immediate instruction
stswx	store string word indexed x-form instruction
stwcx	store word conditional x-form instruction
sync	synchronize instruction
synchronization	The process of arranging storage operations to complete in the order of occurrence.
TAG	MFC command tag
tag group	A group of DMA commands. Each DMA command is tagged with a 5-bit tag group identifier. Software can use this identifier to check or wait on the completion of all queued commands in one or more tag groups. All DMA commands except **getllar**, **putllc**, and **putlluc** are associated with a tag group.
TG	tag parameter
time base	Chip-level time base, as defined in the PowerPC Architecture and in *Cell Broadband Engine Book IV*.
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.

tlbie	translation lookaside buffer invalidate entry instruction
TS	The transfer-size parameter in an MFC command
UME	User mode environment. The instruction set, base command set, storage models, and facilities available to an application programmer.
vector	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 SIMD instructions operate on vector operands. Vectors are also called SIMD operands or packed operands.
vector/SIMD multimedia extension	The SIMD instruction set of the PowerPC Architecture typically supported on a PPE.
virtual address	An address to the virtual-memory space, which is typically much larger than the real address space and includes pages stored on disk. It is translated from an effective address 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.
VPN	Virtual page number. The number of the page in virtual memory.
VS	virtual storage
VSID	virtual segment ID
WIMG bits	Four bits in the page table, also called a page-table entry, which control the processor's accesses to cache and main storage. “W” stands for write through, “I” for cache inhibit, “M” for memory coherence, and “G” for guarded storage.
word	Four bytes.
wrch	write to channel instruction

Index

Numerics

4-byte operation (sndsig), 77
5-bit identifier, 92, 126

A

access
  atomic, 42, 180
  overlapped, MFC, 180
  protection, 53
access exceptions
  local storage, 42
access ordering
  cross-domain storage, 180
  cumulative, 180
  examples, 321
  local storage, 179
  storage, 175
acknowledgment channels
  MFC Write List Stall-and-Notify Tag Acknowledgment Channel, 136
  SPU Write Event Acknowledgment Channel, 161
acronyms, 329
active state, power management, 285
Address Compare Control Register, MFC, 227
address range facility, PPE, 249
address range registers, 249
address registers
  MFC Effective Address High, 86
  MFC Effective Address Low, 87
  MFC Local Storage Address, 85
  MFC Local Storage Address Compare, 229
addresses
  converting (effective to real), 199
  converting (effective to virtual), 199
  converting (virtual to real), 199
  effective
    in address range facility, 250
    list elements, 63
    signal notification registers, 77
  main storage attributes, 31
  overlap, 42
  real
    converting to, 199
  translations, 53, 199, 222
addressing
  local storage, 30
  main storage, 31
alignment errors, 61
atomic access, 180
atomic commands, 60, 314
Atomic Flush Register, MFC, 236
atomic update commands, MFC, 70
atomicity, single-copy, 42
attributes
  main storage, 31
attributes, storage control
  caching inhibited, 44, 177, 218
  guarded, 44, 177, 218
  in real-mode, 218
  memory coherence required, 44, 75–76, 178
  write through required, 44
authentication and decryption master key, 184

B

barrier, 60, 76, 314
barrier commands
  barrier, 60, 76, 314
  defined commands and, 58
  get lock line and reserve command and, 71
  getb, 59, 64, 313
  getbs, 59, 64, 313
  getlb, 59, 65, 313
  global, 77, 314
  list commands and, 63
  MFC atomic update commands and, 70
  MFC synchronization commands, 73
  occurrence of livelock with, 71
  performance note, 76
  put lock line conditional command and, 72
  put lock line unconditional command and, 72
  putb, 58, 66, 312
  putlb, 66
  sndsigb, 60, 78, 314
  tag-specific, 63
barrier, global, 63
barrier, tag-specific, 63
Base Address Register (BP_Base), 79, 293
big-endian byte ordering, 18
binary compatibility with PowerPC, 47
bisled instruction, 153
boundedly undefined behavior, 319
BP_Base, 79, 293
BP_VR, 287

C

cache management instructions, 43
cache models
  overview, 43
cache replacement management, 32, 255
caching inhibited
  storage control attribute, 44, 177, 218

CBEA-compliant processor
  description, 27
  illustration, 28
  organization, 27
CBEA-compliant processor memory map, 294
CBEA-Compliant Processor Version Register, 287
Cell Broadband Engine Architecture, introduction to, 27
channel access facility, SPU, 242
Channel Count Register, SPU, 244
Channel Data Register, SPU, 243
channel input/output, 49
channel instructions, invalid, 113
channel map, SPU, 114, 305
channel names
  MFC_ClassID, 118
  MFC_Cmd, 117
  MFC_EAH, 124
  MFC_EAL, 122
  MFC_LSA, 121
  MFC_RdAtomicStat, 137
  MFC_RdListStallStat, 135, 164
  MFC_RdTagMask, 131
  MFC_RdTagStat, 133
  MFC_Size, 120
  MFC_TagID, 119
  MFC_WrListStallAck, 136, 164
  MFC_WrMSSyncReq, 149
  MFC_WrTagMask, 129
  MFC_WrTagUpdate, 132
  SPU_RdDec, 146
  SPU_RdEventMask, 159
  SPU_RdEventStat, 153
  SPU_RdInMbox, 141
  SPU_RdMachStat, 147
  SPU_RdSigNotify1, 143
  SPU_RdSigNotify2, 144
  SPU_RdSRR0, 148
  SPU_WrDec, 145
  SPU_WrEventAck, 161
  SPU_WrEventMask, 157
  SPU_WrOutIntrMbox, 140
  SPU_WrOutMbox, 139
  SPU_WrSRR0, 148
channels
  command parameter channels, MFC SPU, 116
  MFC Class ID, 118
  MFC Command Opcode, 117
  MFC Command Tag Identification, 119
  MFC Effective Address High, 124
  MFC Effective Address Low or List Address, 122
  MFC Local Storage Address, 121
  MFC Read Atomic Command Status, 137
  MFC Read List Stall-and-Notify Tag Status, 135
  MFC Read Tag-Group Query Mask, 131
  MFC Read Tag-Group Status, 133
  MFC tag-group status, 126
  MFC Transfer Size or List Size, 120
  MFC Write List Stall-and-Notify Tag Acknowledgment, 136
  MFC Write Multisource Synchronization Request, 149
  MFC Write Tag Status Update Request, 132
  MFC Write Tag-Group Query Mask, 129
  reserved, 114, 305
  signalling, 142
  SPU Mailbox, 138
  SPU Read Decrementer, 146
  SPU Read Event Mask, 159
  SPU Read Event Status, 153
  SPU Read Inbound Mailbox, 141
  SPU Read Machine Status, 147
  SPU Read State Save-and-Restore, 148
  SPU Signal Notification 1, 143
  SPU Signal Notification 2, 144
  SPU Write Decrementer, 145
  SPU Write Event Acknowledgment, 161
  SPU Write Event Mask, 157
  SPU Write Outbound Interrupt Mailbox, 140
  SPU Write Outbound Mailbox, 139
  SPU Write State Save-and-Restore, 148
channels, SPU, 114
CIDR, 253
CL (class ID parameter), 62, 311
Class 0 Interrupt Mask Register, 276
Class 0 Interrupt Status Register, 280
class 0 interrupts
  DMA alignment error interrupt, 269
  MFC data segment error interrupt, 269
  MFC data storage error interrupt, 269
  overview, 270
  SPU error interrupt, 269
Class 1 Interrupt Mask Register, 277
Class 1 Interrupt Status Register, 281
class 1 interrupts
  MFC local storage address compare suspend on get interrupt, 269
  overview, 272
Class 2 Interrupt Mask Register, 278
Class 2 Interrupt Status Register, 282
class 2 interrupts
  mailbox interrupt, 269
  overview, 274
  SPU halt instruction trap or single instruction step complete interrupt, 270
  SPU inbound mailbox threshold interrupt, 270
  SPU stop-and-signal instruction trap, 269
  tag-group completion interrupt, 270
class ID parameter (CL), 62, 311
Class ID Register, 253
command classes
  defined, 36
  illegal, 36
  reserved, 36

command completion, determining, 128
Command Error Register, MFC, 231
command exceptions, 61
command issue sequence
  MFC proxy, 88
  MFC SPU, 125
command modifiers
  barrier, 73
  barrier (b), 55, 75
  fence, 73
  fence (f), 55, 75
  list (l), 55
  performance hint (r), 55
  start (s), 55
Command Opcode Channel, MFC, 117
Command Opcode Register, MFC, 81
command parameter channels, MFC SPU, 116, 306
command queue control registers, MFC proxy, 89
command queue, MFC proxy, 55, 91, 175
command queue, MFC SPU, 55, 175
command queues
  MFC proxy, 55, 91, 175
  MFC SPU, 55, 175
Command Status Register, MFC, 90
Command Tag Identification Channel, MFC, 119
commands
  atomic, 314
  atomic update, 70
  barrier, 58–60, 70–73, 76–77, 313–314
  classes, 57
  command modifiers for, 55
  data transfer, 58, 312
  defined, 32, 36, 57, 60
  exceptions, 60
  execution ordering, 177
  get, 64
  get list with fence or barrier, 65
  get with fence or with barrier, 64
  get, table of, 59, 312
  getb, 59, 313
  getbs, 59
  getf, 59
  getfs, 59
  getl, 65
  getlb, 59
  getlf, 59
  getllar, 60, 70, 314
  gets, 59
  illegal, 37
  introduction, MFC, 55
  MFC enforce in-order execution of I/O, 76
  MFC parameters, 62
  MFC, overview, 55
  mfceieio, 60, 70–73, 76, 178, 314, 327
  mfcsync, 60, 70, 72–73, 75, 314, 324
  mnemonics for parameters, 56, 311
  overview, 55
  parameters, MFC, 62
  put, 65
  put with fence or with barrier, 66
  put, table of, 58, 312
  putb, 58, 312
  putbs, 58
  putf, 58
  putfs, 58
  putl, 58
  putlb, 58
  putlf, 58
  putllc, 60, 70–71, 314
  putlluc, 60, 72, 314
  putqlluc, 60, 70, 72, 314
  putr, 58, 66, 312
  putrb, 58
  putrf, 58
  putrl, 58
  putrlb, 58
  putrlf, 58
  puts, 55, 58, 65, 312
  reserved, 37
  sdcrf, 59, 69, 313
  sdcrst, 59, 313
  sdcrt, 59, 67, 313
  sdcrtst, 59, 68, 313
  sdcrz, 59, 68, 313
  See also DMA commands, put commands, MFC commands, and get commands
  send signal, 77
  SL1 data cache, 43
  sndsig, 60, 77, 314
  sndsigb, 60, 314
  sndsigf, 60, 314
  storage control, 59, 67, 313
  synchronization, 60, 70, 72–73, 75, 314
    mfcsync, 71
  touch data cache range, SL1, 67
commands and instructions, defined
  invalid forms, 38
  optional forms, 38
  preferred forms, 37
compatibility
  binary to PowerPC, 47
  PowerPC Architecture (Book I), 47
  PowerPC Architecture (Book II), 48
  PowerPC Architecture (Book III), 197
Configuration Register, SPU, 245
context save and restore registers
  MFC_CNTL, 233
  MFC_Cntl1_ImplRegs, 301
  MFC_CSR_ImplRegs, 193, 301
context save and restore, SPE, 247
control commands, SL1 storage, 59, 313
Control Register, MFC, 233

control registers
  MFC Control, 233
  SPU Privileged Control, 239
conventions and notation, 17
converting addresses (effective to real), 199
converting addresses (effective to virtual), 199
converting addresses (virtual to real), 199
cross-domain storage access ordering, 180
cumulative access ordering, 180
cumulative ordering, 108
cumulative ordering, example, 325

D

DAR, 225
Data Address Register, MFC, 225
data cache range touch command, 67
data range SPRs, 309
Data Storage Interrupt Status Register, 226
data transfer commands, 58, 312
dcbf instruction, 43
dcbst instruction, 43
dcbt instruction, 43
dcbt instruction, X-form, 48
dcbtst instruction, 43
dcbz instruction, 43
debug and power management area, 295
decrementer
  SPU, 145
  SPU Write Decrementer Channel, 145
decrementer channels, SPU, 305
decrementer halt (Dh) bit, 233
decrementer status (Ds) bit, 233
decryption and authentication key, 184
defined behavior for inaccessible SPRs, 319
defined commands, 57
defined commands and instructions
  invalid forms, 38
  optional forms, 38
  preferred forms, 37
defined instructions and commands
  invalid forms, 38
  optional forms, 38
  preferred forms, 37
destructive read, 264
direct external exception, 316
direct external interrupt, 316
direct memory access controller, 52
DMA alignment error interrupt, 269
DMA commands
  get, 64
  get list with fence or with barrier, 65
  get with fence or with barrier, 64
  getl, 64
  getllar, 70
  list elements, introduction, 63
  list size limit, 63
  put, 65–66
  put with fence or with barrier, 66
  putl, 66
  putr, 66
  puts, 65
DMA controller, 52
DMA data transfers, defined, 18
document organization, 15
documentation conventions, 17
domains
  local storage, 30
  main storage, 30
DSISR, 226

E

EAH, 311
EAL, 311
edge-triggered event, 152
Effective Address High Channel, MFC, 124
Effective Address Low or List Address Channel, MFC, 122
Effective Address Low Register, MFC, 87
effective addresses
  list elements, 63
  signal notification registers, 77
  size of address space, 31
Effective Segment ID Register, SLB, 202
eieio instruction, 74–75
endian order, 18
enforce in-order execution of I/O command, MFC (mfceieio), 76
event channels, SPU, 305
event facility, SPU, 150
event support, SPU, 151
events
  edge-triggered, 152
  lock line reservation lost, 170
  MFC list command stall-and-notify, 163
  MFC SPU command queue available, 165
  MFC tag-group status update, 163
  monitored, 150
  multisource synchronization, 172
  privileged attention, 172
  SPU decrementer, 166
  SPU events, introduction, 39
  SPU inbound mailbox available, 166
  SPU outbound interrupt mailbox available, 167
  SPU outbound mailbox available, 168
  SPU signal notification 1 available, 170
  SPU signal notification 2 available, 169
  support illustration, 151

examples of
access ordering, 321
multisource synchronization, 110
exceptions
command, 60
introduction, 38
local storage access, 42
mediated external, 316
extensions
defined behavior for inaccessible SPRs, 319
mediated external exception, 316
multiple concurrent large pages, 318
software management of TLBs, 315
vector/SIMD multimedia, 29, 48, 319
external exceptions
direct, 316
mediated, 316
external interrupt controller, 262
external interrupt definitions, SPU and MFC, 269
external interrupt, direct, 316
external interrupt, mediated, 316
external interrupts, 261

F

facilities
address range, 249
cache replacement management, 32, 255
defined, 32
isolation, SPU, 183
mailbox, 101
MFC, 52
multisource synchronization, MFC, 108
privileged mode, 189
proxy tag-group completion, 92
real-mode address boundary, 218
real-mode storage control, 217
signal notification, SPU, 105
SPU control and status, 96
SPU event, 150
SPU isolation facility, 183
storage control, real-mode, 217
fields and registers, implementation-dependent, 33
fres instruction, A-form, 47
frsqte instruction, A-form, 47
functional components, CBEA, 27

G

get, 59, 64, 312
get commands
get, 59, 64, 312
get list with fence or with barrier option, 65
get with fence or with barrier, 64
getb, 59
getbs, 59, 313
getf, 59, 313
getfs, 59, 313
getl, 64–65
getlb, 59, 313
getlf, 59, 313
getllar, 70
getllar (get lock-line and reserve), 70
gets, 59, 312
introduction, 64
table of, 59, 312
getl, 64
getb, 64
getbs, 59, 64, 313
getf, 59, 64, 313
getfs, 59, 64, 313
getl, 65
getlb, 59, 65, 313
getlf, 59, 65, 313
getllar, 60, 70, 314
gets, 59, 64, 312
global barrier command, 63
guarded pages, in list transfers, 63
guarded, storage control attribute, 44, 177, 218

H

Harvard-style cache, 43
hypervisor
causing mediated external interrupts, 316
use with multiple operating systems, 189

I

icbi instruction, 43
IDEA (implementation-dependent expansion area), 293, 295
Identification Register, SPU, 291
IIC (internal interrupt controller)
area, 295
CBEA organization, 27
description, 30
interrupts sent to, 262
memory map, 293, 303
thread control block, 295
illegal instructions, 37
implementation-dependent expansion area (IDEA), 293, 295
implementation-dependent fields
overview, 33
implementation-dependent fields and registers, 33

implementation-dependent registers
  Base Address Register, 79, 293
  BP_Base, 293
  L2_RMT_Data, 195
  L2_RMT_Index, 195
  MFC Atomic Flush Register, 190, 236
  MFC Context Save and Restore, 193
  MFC Real-Mode Address Boundary Register, 218
  MFC_Atomic_Flush, 190, 236
  MFC_Cntl1_ImplRegs, 193
  MFC_CSR_ImpRegs, 193
  MFC_RMAB, 218
  MFC_TLB_RMT_Data, 191
  MFC_TLB_RMT_Index, 191
  overview, 33
  privilege 1, 192, 300
  privilege 2, 194
  PV2_ImplRegs, 194
  RMT Data Register, 195, 258
  RMT Index Register, 195, 257
  RMT_Data, 258
  RMT_Index, 257
  SPE_MMU_ImplRegs, 191
  SPE_RMT_ImplRegs, 191
  SPU cache hardware registers, 190
  SPU_Cache_ImpRegs, 190
inaccessible SPRs
  defined behavior for, 319
Inbound Mailbox Register, SPU, 103
incompatibilities
  PowerPC Architecture (Book I), 47
  PowerPC Architecture (Book II), 48
  PowerPC Architecture (Book III), 197
index generation
  replacement management table, 256
  RMT example, 257
Index Hint Register, TLB, 209
Index Register, SLB, 201
Index Register, TLB, 210
instruction caused interrupt, 261
instruction classes
  defined, 36
  illegal, 36
  reserved, 36
instruction range SPRs, 309
instructions
  bisled, 153
  branch-conditional, 47
  cache management
    dcbf, 43
    dcbst, 43
    dcbt, 43
    dcbtst, 43
    dcbz, 43
    icbi, 43
  dcbf, 43
  dcbst, 43
  dcbt, 43
  dcbt, X-form, 48
  dcbtst, 43
  dcbz, 43
  defined, 32
  defined class, 36
  eieio, 74–75
  fres, 47
  fres, A-form, 47
  frsqte, 47
  frsqte, A-form, 47
  icbi, 43
  illegal class, 37
  invalid channel, 113
  ld, 291
  ldarx, 70
  lwarx, 70
  lwsync, 74
  mfspr, 209–212
  mtspr, 210–212
  reserved class, 37
  SLB
    slbia, 200
    slbie, 200
    slbmfee, 200
    slbmfev, 200
    slbmte, 200
  slbia, 200
  slbie, 200
  slbmfee, 200
  slbmfev, 200
  slbmte, 200
  stdcx., 70
  stwcx., 70
  sync, 74
  TLB
    tlbia, 216
    tlbie, 208, 214
    tlbiel, 214
  tlbia, 216
  tlbie, 208, 214
  tlbiel, 214
instructions and commands, defined
  invalid forms, 38
  optional forms, 38
  preferred forms, 37
INT_CPL, 268
INT_Mask_class0, 276
INT_Mask_class1, 277
INT_Mask_class2, 278
INT_Stat_class0, 280
INT_Stat_class1, 281
INT_Stat_class2, 282

internal interrupt controller (IIC)
  area, 295
  CBEA organization, 27
  description, 30
  interrupts sent to, 262
  memory map, 293, 303
  registers, 263
  thread control block, 295
interprocessor interrupts, 267
interrupt classes, 261
  application (class 2), 269
  error (class 0), 269
  translation (class 1), 269
interrupt control block, 263
interrupt controller
  external, 262
  internal, 262
Interrupt Current Priority Level Register, 268
Interrupt Generation Port Register, 267
interrupt generation process
  class 0, MFC, illustration, 271
  class 1, MFC, illustration, 273
  class 2, MFC, illustration, 275
  SPU and MFC, 270
Interrupt Pending Port Register, 263
interrupt register names
  INT_CPL, 268
  INT_Generation, 267
  INT_Mask_class0, 276
  INT_Mask_class1, 277
  INT_Mask_class2, 278
  INT_Pending_D, 263
  INT_Pending_NonD, 263
  INT_Route, 283
  INT_Stat_class0, 280
  INT_Stat_class1, 281
  INT_Stat_class2, 282
interrupt registers
  Class 0 Interrupt Mask, 276
  Class 0 Interrupt Status, 280
  Class 1 Interrupt Mask, 277
  Class 1 Interrupt Status, 281
  Class 2 Interrupt Mask, 278
  Class 2 Interrupt Status, 282
  Interrupt Current Priority Level Register, 268
  Interrupt Generation Port, 267
  Interrupt Pending Port, 263
  Interrupt Routing, 283
Interrupt Routing Register, 283
interrupt status registers, overview, 279
interrupt-related channels, SPU, 148
interrupts
  class 0, 270
  class 1, 272
  class 2, 274
  class definitions, MFC and SPU, 269
  external, 261
  external interrupt definitions, SPU and MFC, 269
  instruction caused, 261
  interprocessor, 267
  system caused, 261
invalid channel instruction, 113
invalid instructions, 38
Invalidate All Register, SLB, 206
Invalidate All Register, TLB, 216
Invalidate Entry Register, SLB, 205
Invalidate Entry Register, TLB, 214
invalidate selector (IS) field, 315
IS (invalidate sector) field, 315
isolated state, power management, 285
isolation facility, SPU, 183

L

LA (list local storage address) parameter, 63, 311
large page sizes, 212
large pages
  multiple concurrent, 318
ld instruction, 291
ldarx instruction, 70
LEAL (List Effective Address Low field), 64
LEAL (list effective address low) parameter, 63, 311
legal notice, 2
list commands, 63, 135
list effective address low (LEAL) parameter, 63, 311
List Effective Address Low field (LEAL), 64
list element parameters
  LA, 63
  LEAL, 63
  LTS, 63
list elements, 63, 135
list size (LSZ) parameter, 311
little-endian byte ordering, 18
load control bit, TLB, 315
load-store
  architecture, 49
  operations, 53
local storage
  addressing, 30
  defined, 30
  MFC Local Storage Address Channel, 121
  MFC Local Storage Address Register, 85
  SPU Local Storage Limit Register, 241
  SPU model, 41
local storage access
  exceptions, 42
  introduction, 42
  mapping requirements, 42
  ordering, 179
local storage address (LSA) parameter, 311
Local Storage Address Channel, MFC, 121

local storage address parameter (LSA), 62
Local Storage Address Register, MFC, 85
local storage domain, 30
Local Storage Limit Register, SPU, 241
lock line reservation lost event, 170
Logical Partition ID Register, MFC, 223
logical partitioning, operating environment, 189
LSA (local storage address parameter), 62, 311
LSZ (list size parameter), 62, 311
LTS (list element transfer size), 63, 311
lwarx instruction, 70
lwsync instruction, 74

M

mailbox channels, SPU, 138
mailbox facility
  overview, 101
  SPU Inbound Mailbox Register, 103
  SPU mailbox channels, 138
  SPU Mailbox Status Register, 104
  SPU Outbound Interrupt Mailbox, 237
  SPU Outbound Mailbox Register, 102
  SPU Read Inbound Mailbox Channel, 141
  SPU Write Outbound Interrupt Mailbox Channel, 140
  SPU_In_Mbox, 103
  SPU_Mbox_Stat, 104
  SPU_Out_Mbox, 102
  SPU_RdInMbox Channel, 141
  SPU_WrOutIntrMbox, 140
  SPU_WrOutMbox, 139
mailbox interrupt, 269
Mailbox Status Register, SPU, 104
mailboxes, SPU, 306
main storage
  addressing, 31
  attributes, 31
  defined, 30
  sharing, 31
main storage addressing, 31
many-to-one signalling environment, 77
mapping requirements
  local storage access, 42
mapping, TLB, 208
master key, authentication and decryption, 184
mediated external exception, 316
mediated external exception extension, 316
mediated external interrupt, 316
memory coherence
  defined, 43
memory coherence required
  storage control attribute, 44, 75–76, 178
memory flow controller (MFC)
  block diagram, 51
  CBEA organization, 27
  description, 29
  overview, 51
  typical, 52
memory management unit (MMU), 53
memory maps
  CBEA-compliant processor, 294
  internal interrupt controller, 263, 303
  introduction, 293
  PPE privilege 1, 195, 302
  SPE privilege 1, 190, 298
  SPE privilege 2, 193, 301
  SPE problem state, 79, 296
MFC (memory flow controller)
  block diagram, 51
  CBEA organization, 27
  description, 29
  overview, 51
  typical, 52
MFC Address Compare Control Register, 227
MFC atomic commands
  getllar, 314
  putllc, 314
  putlluc, 314
  putqlluc, 314
MFC Atomic Flush Register, 236
MFC Command Error Register, 231
MFC Command Opcode Channel, 117
MFC Command Opcode Register, 81
MFC command parameters, 62
MFC Command Status Register, 90
MFC Command Tag Identification Channel, 119
MFC commands
  barrier, 60
  mfceieio, 60, 70–73, 76, 178, 314, 327
  mfcsync, 60, 70–73, 75, 314
  overview, 55
  proxy, 55
  See also commands
  SPU, 55
MFC Control Register, 233
MFC Data Address Register, 225
MFC data segment error interrupt, 269
MFC data storage error interrupt, 269
MFC DMA commands, 55
  See also commands
MFC DSI register names
  MFC_DSIPR, 232
  MFC_LSACR, 229
  MFC_LSCRR, 230
MFC Effective Address High Channel, 124
MFC Effective Address Low or List Address Channel, 122
MFC Effective Address Low Register, 87
MFC facilities, *see* facilities
MFC Interrupt Routing Register, 283
MFC list command stall-and-notify event, 163
MFC Local Storage Address Channel, 121

MFC local storage address compare suspend on get interrupt, 269
MFC Local Storage Address Register, 85
MFC Logical Partition ID Register, 223
MFC multisource synchronization channels, 305
MFC multisource synchronization facility
  example, 110
  MFC Multisource Synchronization Register, 109
  overview, 108
MFC Multisource Synchronization Register, 109
MFC overlapped access, 180
MFC proxy command issue sequence, 88
MFC proxy command queue, 91, 175
MFC proxy command queue status and control registers, 89
MFC proxy commands, 55
MFC Queue Status Register, 91
MFC Read Atomic Command Status Channel, 137
MFC Read List Stall-and-Notify Tag Status Channel, 135
MFC Read Tag-Group Query Mask Channel, 131
MFC Read Tag-Group Status Channel, 133
MFC Real-Mode Address Boundary Register, 218
MFC register names
  MFC_ACCR, 227
  MFC_Atomic_Flush, 236
  MFC_CER, 231
  MFC_ClassID, 82
  MFC_CMD, 81
  MFC_CMDStatus, 90
  MFC_CNTL, 233
  MFC_DAR, 225
  MFC_DSIPR, 232
  MFC_DSISR, 226
  MFC_EAH, 86
  MFC_EAL, 87
  MFC_LPID, 223
  MFC_LSA, 85
  MFC_LSACR, 229
  MFC_MSSync, 109
  MFC_QStatus, 91
  MFC_RMAB, 218
  MFC_Size, 84
  MFC_SR1, 221
  MFC_Tag, 83
  MFC_VR, 290
MFC registers
  MFC Address Compare Control, 227
  MFC Class ID, 82
  MFC Command Error, 231
  MFC Command Opcode, 81
  MFC Command Status, 90
  MFC Command Tag, 83
  MFC Data Address, 225
  MFC Data Storage Interrupt Pointer, 232
  MFC Data Storage Interrupt Status, 226
  MFC Effective Address High, 86
  MFC Effective Address Low, 87
  MFC Local Storage Address, 85
  MFC Local Storage Address Compare, 229
  MFC Local Storage Compare Result, 230
  MFC Logical Partition ID, 223
  MFC Multisource Synchronization, 109
  MFC Queue Status, 91
  MFC State Register One, 221
  MFC Storage Description, 224
  MFC Transfer Size, 84
  MFC Version, 290
MFC SPU command parameter channels, 116, 306
MFC SPU command queue available event, 165
MFC SPU commands, 55
MFC SPU queue, 175
MFC State Register One, 221
MFC Storage Description Register, 224
MFC synchronization commands
  barrier, 314
  mfceieio, 314
  mfcsync, 314
  sndsig, 314
  sndsigb, 314
  sndsigf, 314
MFC tag status channels, 306
MFC tag-group status update event, 163
MFC Transfer Size or List Size Channel, 120
MFC Version Register, 290
MFC Write List Stall-and-Notify Tag Acknowledgment Channel, 136
MFC Write Multisource Synchronization Request Channel, 149
MFC Write Tag Status Update Request Channel, 132
MFC Write Tag-Group Query Mask Channel, 129
MFC_ACCR Register, 227
MFC_Atomic_Flush Register, 236
MFC_CER Register, 231
MFC_ClassID Channel, 118
MFC_ClassID Register, 82
MFC_Cmd Channel, 117
MFC_CMD Register, 81
MFC_CMDStatus Register, 90
MFC_CNTL Register, 233
MFC_DAR Register, 225
MFC_DSIPR Register, 232
MFC_DSISR Register, 226
MFC_EAH Channel, 124
MFC_EAH Register, 86
MFC_EAL Channel, 122
MFC_EAL Register, 87
MFC_LPID Register, 223
MFC_LSA Channel, 121
MFC_LSA Register, 85
MFC_LSCAR Register, 229
MFC_LSCRR Register, 230
MFC_MSSync Register, 109

MFC_QStatus Register, 91
MFC_RdAtomicStat Channel, 137
MFC_RdListStallStat Channel, 135
MFC_RdTagMask Channel, 131
MFC_RdTagStat Channel, 133
MFC_RMAB Register, 218
MFC_SDR Register, 224
MFC_Size Channel, 120
MFC_Size Register, 84
MFC_SR1 Register, 221
MFC_Tag Register, 83
MFC_TagID Channel, 119
MFC_VR Register, 290
MFC_WrListStallAck Channel, 136
MFC_WrMSSyncReq Channel, 149
MFC_WrTagMask Channel, 129
MFC_WrTagUpdate Channel, 132
mfceieio command, 60, 70–73, 76, 178, 314
mfceieio command, example, 327
mfcsync command, 60, 70–73, 75, 314
mfcsync command, example, 324
mfspr instruction, 209–212
MMIO interface, 52
models
   cache, 43
   Harvard-style cache, 43
   SPU local storage, 41
monitored events, 150
mtspr instruction, 210–212
multimedia extensions
   vector/SIMD, 29
multiple concurrent large pages, 318
multisource synchronization channels, MFC, 305
multisource synchronization event, 172
multisource synchronization facility, MFC, 108
Multisource Synchronization Register, MFC, 109

N

Next Program Counter Register, SPU, 99
nondestructive read, 264, 279

O

one-to-one signalling environment, 77
operating environment
   logical partitioning, 189
operating states, SPU, 185
optional fields, 38
optional instructions, 38
ordering, access
   cross-domain storage, 180
   cumulative, 180
   local storage, 179
ordering, command execution, 177
organization, privileged mode facilities, 189
Outbound Interrupt Mailbox Register, SPU, 237
Outbound Mailbox Register, SPU, 102
overlap, addresses, 42
overlapped access, MFC, 180

P

page
   defined, 41
   large size, 212
   real and virtual, 41
parameters, MFC command
   CL, 311
   EAH, 311
   EAL, 311
   LA, 311
   LEAL, 311
   LSA, 311
   LSZ, 311
   LTS, 311
   overview, 62
   TG, 311
   TS, 311
pause state, power management, 285
Pending Event Register, SPU, 157
PMD (power management and debug) area, 293, 295
power area(p), 295
power management, 285
power management and debug (PMD) area, 293, 295
PowerPC instructions
   ldarx, 70
   lwarx, 70
   lwsync, 74
   mfspr, 209, 211
   mtspr, 210–211
   slbia, 200
   slbie, 200
   slbmfee, 200
   slbmfev, 200
   slbmte, 200
   stdcx., 70
   stwcx., 70
   sync, 74
   tlbia, 216
   tlbie, 208, 214
   tlbiel, 214
PowerPC Processor Element (PPE)
   area, 293
   CBEA organization, 27
   compatibility, 47–48
   description, 29
   special purpose register map, 309

PPE (PowerPC Processor Element)
  area, 293
  CBEA organization, 27
  compatibility, 47–48
  description, 29
  special purpose register map, 309
PPE (PowerPC Processor Element) special purpose registers, 309
PPE privilege 1 memory map, 302
PPE Processor Version Register, 288
preemptive context switching, 247
privilege 1 register names
  INT_Mask_class0, 276
  INT_Mask_class1, 277
  INT_Mask_class2, 278
  INT_Route, 283
  INT_Stat_class0, 280
  INT_Stat_class1, 281
  INT_Stat_class2, 282
  MFC_ACCR, 227
  MFC_Atomic_Flush, 236
  MFC_CER, 231
  MFC_DAR, 225
  MFC_DSIPR, 232
  MFC_DSISR, 226
  MFC_LPID, 223
  MFC_LSACR, 229
  MFC_LSCRR, 230
  MFC_RMAB, 218
  MFC_SDR, 224
  MFC_SR1, 221
  MFC_VR, 290
  PPEPV_ImplRegs, 195
  RMT_Data, 258
  RMT_Index, 257
  SPU_ID, 291
  SPU_VR, 289
  TLB_Index, 210
  TLB_Index_Hint, 209
  TLB_Invalidate_All, 216
  TLB_Invalidate_Entry, 214
  TLB_RPN, 212
  TLB_VPN, 211
privilege 1 registers
  Class 0 Interrupt Mask, 276
  Class 0 Interrupt Status, 280
  Class 1 Interrupt Mask, 277
  Class 1 Interrupt Status, 281
  Class 2 Interrupt Mask, 278
  Class 2 Interrupt Status, 282
  implementation-dependent, 192, 300
  Interrupt Routing, 283
  introduction to, 189
  MFC Address Compare Control, 227
  MFC Atomic Flush, 236
  MFC Command Error, 231
  MFC Data Address, 225
  MFC Data Storage Interrupt Pointer, 232
  MFC Data Storage Interrupt Status, 226
  MFC Local Storage Address Compare, 229
  MFC Local Storage Compare Result, 230
  MFC Logical Partition ID, 223
  MFC Real-Mode Address Boundary, 218
  MFC State Register One, 221
  MFC Storage Description, 224
  MFC Version, 290
  PPE Processor Version, 288
  RMT Data, 258
  RMT Index, 257
  SPU Identification, 291
  SPU Version, 289
  TLB Index, 210
  TLB Index Hint, 209
  TLB Invalidate All, 216
  TLB Invalidate Entry, 214
  TLB Real Page Number, 212
  TLB Virtual Page Number, 211
privilege 2 register names
  MFC_CNTL, 233
  MFC_Cntl1_ImplRegs, 301
  MFC_CSR_ImplRegs, 193, 301
  PV2_ImplRegs, 194, 302
  SLB_ESID, 202
  SLB_Index, 201
  SLB_Invalidate_All, 206
  SLB_VSID, 203
  SPU_Cfg, 245
  SPU_ChnlCnt, 244
  SPU_ChnlData, 243
  SPU_ChnlIndex, 242
  SPU_LSLR, 241
  SPU_Out_Intr_Mbox, 237
  SPU_PrivCntl, 239
privilege 2 registers
  MFC Control, 233
  SLB Effective Segment ID, 202
  SLB Index, 201
  SLB Invalidate All, 206
  SLB Invalidate Entry, 205
  SPU Channel Count, 244
  SPU Channel Data, 243
  SPU Channel Index, 242
  SPU Configuration, 245
  SPU Local Storage Limit, 241
  SPU Outbound Interrupt Mailbox, 237
  SPU Privileged Control, 239
privileged area, SPE, 294
privileged attention event, 172
Privileged Control Register, SPU, 239
privileged mode environment, 27

privileged mode facility
introduction to, 189
organization, 189
privileged state
SPE privilege 1 memory map, 190, 298, 301
SPE privilege 2 memory map, 193
privileged state area, 294
problem state, 294
problem state memory map, 79
Processor Version Register, CBEA-Compliant, 287
protection
access, 53
proxy command issue sequence, MFC, 88
proxy command queue status and control registers, MFC, 89
proxy command queue, MFC, 91, 175
proxy register names
Prxy_QueryMask, 94
Prxy_QueryType, 93
Prxy_TagStatus, 95
proxy registers
Proxy Tag-Group Query Mask, 94
Proxy Tag-Group Query Type, 93
Proxy Tag-Group Status, 95
proxy tag-group completion facility, 92
Proxy Tag-Group Query Mask Register, 94
Proxy Tag-Group Status Register, 95
Prxy_Query Mask Register, 94
Prxy_Query Type Register, 93
Purge (Pc) bit, 233
purge sequence (Pc and Ps), 234
put, 58, 65, 312
put commands
overview, 65
put, 58, 65–66, 312
putb, 58, 312
putbs, 58, 312
putf, 58, 312
putfs, 58, 312
putl, 58, 66, 312
putlb, 58, 312
putlf, 58, 312
putllc, 60, 70–71, 314
putlluc, 60, 72, 314
putqlluc, 60, 70, 72, 314
putr, 58, 66, 312
putrb, 58, 312
putrf, 58, 312
putrl, 58, 312
putrlb, 58, 312
putrlf, 58, 312
puts, 55, 58, 65, 312
table of, 58, 312
putb, 58, 66, 312
putbs, 58, 66, 312
putf, 58, 66, 312
putfs, 58, 66, 312
putl, 58, 66, 312
putlb, 58, 66, 312
putlf, 58, 66, 312
putllc, 60, 71, 314
putlluc, 60, 72, 314
putqlluc, 60, 72, 314
putr, 58, 312
putrb, 58, 312
putrf, 58, 312
putrl, 58, 312
putrlb, 58, 312
putrlf, 58, 312
puts, 55, 58, 65, 312
PV1_ImplRegs, 192, 300
PV2_ImplRegs, 194, 302
PVR, 288

Q

Queue Status Register, MFC, 91
queues
MFC proxy command, 175
MFC SPU command, 175
queuing
MFC command from SPU, 125

R

Range Mask Register, 252
range registers, 249
Range Start Register, 251
Read Atomic Command Status Channel, MFC, 137
read channels
MFC Read Atomic Command Status, 137
MFC Read List Stall-and-Notify Tag Status, 135
MFC Read Tag-Group Query Mask, 131
MFC Read Tag-Group Status, 133
SPU Read Decrementer, 146
SPU Read Event Mask, 159
SPU Read Event Status, 153
SPU Read Inbound Mailbox Channel, 141
SPU Read Machine Status, 147
SPU Read State Save-and-Restore, 148
Read Decrementer Channel, SPU, 146
Read Event Mask Channel, SPU, 159
Read Event Status Channel, SPU, 153
Read Inbound Mailbox Channel, SPU, 141
Read List Stall-and-Notify Tag Status Channel, MFC, 135
Read Machine Status Channel, SPU, 147
Read State Save-and-Restore Channel, SPU, 148
Read Tag-Group Query Mask Channel, MFC, 131
Read Tag-Group Status Channel, MFC, 133
reads, destructive, 264

reads, nondestructive, 264, 279
real address
  converting, 199
  local storage access to, 42
  local storage addressing, 30
  main storage addressing, 31
  size of address space, 31
real addressing mode, 217, 251
real page, 41
Real Page Number Register, TLB, 212
real-mode address boundary facility, MFC, 218
Real-Mode Address Boundary Register, MFC, 218
real-mode storage control facilities, 217
register names
  BP_Base, 293
  BP_VR, 287
  CIDR, 253
  INT_CPL, 268
  INT_Generation, 267
  INT_Mask_class0, 276
  INT_Mask_class1, 277
  INT_Mask_class2, 278
  INT_Pending_D, 263
  INT_Pending_NonD, 263
  INT_Route, 283
  INT_Stat_class0, 280
  INT_Stat_class1, 281
  INT_Stat_class2, 282
  MFC_ACCR, 227
  MFC_Atomic_Flush, 236
  MFC_CER, 231
  MFC_ClassID, 82
  MFC_CMD, 81
  MFC_CMDStatus, 90
  MFC_CNTL, 233
  MFC_Cntl1_ImplRegs, 301
  MFC_CSR_ImplRegs, 193, 301
  MFC_DAR, 225
  MFC_DSIPR, 232
  MFC_DSISR, 226
  MFC_EAH, 86
  MFC_EAL, 87
  MFC_LPID, 223
  MFC_LSA, 85
  MFC_LSACR, 229
  MFC_LSCRR, 230
  MFC_MSSync, 109
  MFC_QStatus, 91
  MFC_RMAB, 218
  MFC_SDR, 224
  MFC_Size, 84
  MFC_SR1, 221
  MFC_Tag, 83
  MFC_TLB_RMT_Index, 257
  MFC_VR, 290
  PPEPV_ImplRegs, 195
  Prxy_QueryMask, 94
  Prxy_QueryType, 93
  Prxy_TagStatus, 95
  PV2_ImplRegs, 194, 302
  PVR, 288
  RMR, 252
  RMT, 255
  RMT_Data, 258
  RMT_Index, 257
  RSR, 251
  SLB_ESID, 202
  SLB_Index, 201
  SLB_Invalidate_All, 206
  SLB_Invalidate_Entry, 205
  SLB_VSID, 203
  SPU_Cache_ImplRegs, 190
  SPU_Cfg, 245
  SPU_ChnlCnt, 244
  SPU_ChnlData, 243
  SPU_ChnlIndex, 242
  SPU_ID, 291
  SPU_In_Mbox, 103
  SPU_LSLR, 241
  SPU_Mbox_Stat, 104
  SPU_NPC, 99
  SPU_Out_Intr_Mbox, 237
  SPU_Out_Mbox, 102
  SPU_PrivCntl, 239
  SPU_RunCntl, 96
  SPU_Sig_Notify_1, 106
  SPU_Sig_Notify_2, 107
  SPU_Status, 97
  SPU_VR, 289
  TLB_Index, 210
  TLB_Index_Hint, 209
  TLB_Invalidate_All, 216
  TLB_Invalidate_Entry, 214
  TLB_RPN, 212
  TLB_VPN, 211
register representation, 18
registers
  Address Range, 249
  Base Address Register, 293
  CBEA-Compliant Processor Version, 287
  Class 0 Interrupt Mask, 276
  Class 0 Interrupt Status, 280
  Class 1 Interrupt Mask, 277
  Class 1 Interrupt Status, 281
  Class 2 Interrupt Mask, 278
  Class 2 Interrupt Status, 282
  Class ID, 253
  command queue control, proxy, 89
  Interrupt Current Priority Level, 268
  Interrupt Generation Port, 267
  Interrupt Pending Port, 263
  Interrupt Routing, 283

MFC Address Compare Control, 227
MFC Atomic Flush, 236
MFC Class ID, 82
MFC Command Error, 231
MFC Command Opcode, 81
MFC Command Status, 90
MFC Command Tag, 83
MFC Control, 233
MFC Data Address, 225
MFC Data Storage Interrupt Pointer, 232
MFC Data Storage Interrupt Status, 226
MFC Effective Address High, 86
MFC Effective Address Low, 87
MFC Local Storage Address, 85
MFC Local Storage Address Compare, 229
MFC Local Storage Compare Result, 230
MFC Logical Partition ID, 223
MFC Multisource Synchronization, 109
MFC Queue Status, 91
MFC Real-Mode Address Boundary, 218
MFC State Register One, 221
MFC Status Interrupt, 226
MFC Storage Description, 224
MFC Transfer Size, 84
MFC Version, 290
PPE Processor Version, 288
problem state memory map, 79, 296
Proxy Tag-Group Query Mask, 94
Proxy Tag-Group Query Type, 93
Proxy Tag-Group Status Register, 95
Range Mask, 252
Range Start, 251
RMT Data, 258
RMT Index, 257
SLB Effective Segment ID, 202
SLB Index, 201
SLB Invalidate All, 206
SLB Invalidate Entry, 205
SLB Virtual Segment ID, 203
special purpose, 209–210
SPU Channel Count, 244
SPU Channel Data, 243
SPU Channel Index, 242
SPU Configuration, 245
SPU Identification, 291
SPU Inbound Mailbox, 103
SPU Local Storage Limit, 241
SPU Mailbox Status, 104
SPU Next Program Counter, 99
SPU Outbound Interrupt Mailbox, 237
SPU Outbound Mailbox, 102
SPU Pending Event, 157
SPU Privileged Control, 239
SPU Run Control, 96
SPU Signal Notification 1, 106
SPU Signal Notification 2, 107
SPU Status, 97
SPU Version, 289
standard representation, 18
TLB Index, 210
TLB Index Hint, 209
TLB Invalidate All, 216
TLB Invalidate Entry, 214
TLB Real Page Number, 212
TLB Virtual Page Number, 211

related publications, 16
replacement management
  cache, 32, 255
replacement management table (RMT)
  CBEA organization, 27
  example, 255
  Replacement Management Table Register, 255
  SPU component, 27
replacement management table (RMT) SPRs, 309
Replacement Management Table Data Register, 258
Replacement Management Table Index Register, 257
requesters, resource allocation management, 259
required features (for CBEA)
  optional in *PowerPC Architecture (Book I and Book II)*, 47
  optional in the *PowerPC Architecture (Book III)*, 197
reserved bits
  handling, 33
  preserving state, 33
reserved channels, 114, 305
reserved fields, 32
reserved registers, setting to zero, 33
resource allocation group (RAG), 259
resource allocation management, 259
restart (R) bit, 233–234
restore and save, context, SPE, 247
restrictions
  SPU, 49
revision log, 19
RMR, 252
RMT (replacement management table)
  CBEA organization, 27
  example, 255
  Replacement Management Table Register, 255
  SPU component, 27
RMT Data Register, 258
RMT register names
  MFC_TLB_RMT_Index, 257
RMT registers
  Index Mask, 256
  Index Off, 256
  RMT Data, 258
  RMT Index, 257
RMT_Index, 257

RSR, 251
Run Control, SPU, 96

S

S (Stall-and-Notify bit), 63
save and restore, context, SPE, 247
sdcrf, 59, 69, 313
sdcrst, 59, 69, 313
sdcrt, 59, 67, 313
sdcrtst, 59, 68, 313
sdcrz, 59, 68, 313
segment lookaside buffer management, PPE, 200
send signal command, 77
shared storage, 45
signal control registers, 77
Signal Notification 1 Channel, SPU, 143
Signal Notification 1 Register, SPU, 106
Signal Notification 2 Channel, SPU, 144
Signal Notification 2 Register, SPU, 107
signal notification channels, SPU, 305
signal notification facility, SPU, 105
signal notification registers
  SPU Signal Notification 1, 106
  SPU Signal Notification 2, 107
signal, send command, 77
signalling channels, 142
signalling environment
  many-to-one, 77
  one-to-one, 77
SIMD (single instruction, multiple data), 29, 49
single instruction step mode, 239
single instruction, multiple data (SIMD), 29, 49
single-copy atomicity, 42
size of list, 63
sizes of main storage address space, 31
SL1 data cache commands
  description, 43
  sdcrf, 69
  sdcrst, 68
  sdcrt, 67
  sdcrtst, 68
  sdcrz, 68
SL1 storage control, 59, 313
SL1 storage control commands
  sdcrf, 59, 313
  sdcrst, 59, 313
  sdcrt, 59, 313
  sdcrtst, 59, 313
  sdcrz, 59, 313
SLB Effective Segment ID Register, 202
SLB Index Register, 201
SLB instructions
  slbia, 200
  slbie, 200
  slbmfee, 200
  slbmfev, 200
  slbmte, 200
SLB Invalidate All Register, 206
SLB Invalidate Entry Register, 205
SLB mapping, 200
SLB register names
  SLB_ESID, 202
  SLB_Index, 201
  SLB_Invalidate_All, 206
  SLB_Invalidate_Entry, 205
  SLB_VSID, 203
SLB registers
  SLB Effective Segment ID, 202
  SLB Index, 201
  SLB Invalidate All, 206
  SLB Invalidate Entry, 205
  SLB Virtual Segment ID, 203
SLB Virtual Segment ID Register, 203
SLB_ESID Register, 202
SLB_Index Register, 201
SLI (state lost and isolated), power management, 285
slow state, power management, 285
sndsig, 60, 77, 314
sndsigb, 60, 78, 314
sndsigf, 60, 78, 314
software management of TLBs, 315
SPE local storage, 294
SPE privilege 1 memory map, 298
SPE privilege 2 memory map, 301
SPE privileged area, 294
SPE problem state memory map, 296
special purpose register map, PPE, 309
special purpose registers
  PPE, 309
  TLB Index Hint (TLB_Index_Hint), 209
  TLB Index Register (TLB_Index), 210
SPU (synergistic processor unit)
  CBEA organization, 27
  description, 29
  overview, 49
SPU Channel Count Register, 244
SPU Channel Data Register, 243
SPU channel map, 114, 305
SPU channels, 114
SPU command parameter channels, MFC, 116, 306
SPU configuration, 245
SPU Configuration Register, 245
SPU control and status facilities, 96
SPU decrementer, 145
SPU decrementer channels, 305
SPU decrementer event, 166
SPU error interrupt, 269
SPU event channels, 305
SPU event facility, 150
SPU event support

illustration, 151
SPU events
lock line reservation lost, 170
MFC list command stall-and-notify, 163
MFC SPU command queue available, 165
MFC tag-group status update event, 163
multisource synchronization, 172
privileged attention, 172
SPU decrementer, 166
SPU inbound mailbox available event, 166
SPU outbound interrupt mailbox available, 167
SPU outbound mailbox available, 168
SPU signal notification 1 available, 170
SPU signal notification 2 available, 169
support
illustration, 151
SPU events, introduction, 39
SPU halt instruction trap or single instruction step complete interrupt, 270
SPU Identification Register, 291
SPU inbound mailbox available event, 166
SPU Inbound Mailbox Register, 103
SPU inbound mailbox threshold interrupt, 270
SPU interrupt-related channels, 148
SPU isolation facility, 183
SPU Local Storage Limit Register, 241
SPU local storage problem state area, 293
SPU mailbox channels, 138
SPU Mailbox Status Register, 104
SPU mailboxes, 115, 306
SPU Next Program Counter Register, 99
SPU operating states, 185
SPU outbound interrupt mailbox available event, 167
SPU Outbound Interrupt Mailbox Register, 237
SPU outbound mailbox available event, 168
SPU Outbound Mailbox Register, 102
SPU Pending Event Register, 157
SPU Privileged Control Register, 239
SPU Read Decrementer Channel, 146
SPU Read Event Mask Channel, 159
SPU Read Event Status Channel, 153
SPU Read Inbound Mailbox Channel, 141
SPU Read Machine Status Channel, 147
SPU Read State Save-and-Restore Channel, 148
SPU register names
SPU_Cache_ImplRegs, 190
SPU_Cfg, 245
SPU_ChnlCnt, 244
SPU_ChnlData, 243
SPU_ChnlIndex, 242
SPU_ID, 291
SPU_In_Mbox, 103
SPU_LSLR, 241
SPU_Mbox_Stat, 104
SPU_NPC, 99
SPU_Out_Intr_Mbox, 237
SPU_Out_Mbox, 102
SPU_PrivCntl, 239
SPU_RunCntl, 96
SPU_Sig_Notify_1, 106
SPU_Sig_Notify_2, 107
SPU_Status, 97
SPU_VR, 289
SPU registers
SPU Channel Count, 244
SPU Channel Data, 243
SPU Channel Index, 242
SPU Configuration Register, 245
SPU Identification, 291
SPU Inbound Mailbox, 103
SPU Local Storage Limit, 241
SPU Mailbox Status, 104
SPU Next Program Counter, 99
SPU Outbound Interrupt Mailbox, 237
SPU Outbound Mailbox, 102
SPU Privileged Control, 239
SPU Run Control, 96
SPU Signal Notification 1, 106
SPU Signal Notification 2, 107
SPU Status, 97
SPU Version, 289
SPU signal notification 1 available event, 170
SPU Signal Notification 1 Channel, 143
SPU Signal Notification 1 Register, 106
SPU signal notification 2 available event, 169
SPU Signal Notification 2 Channel, 144
SPU Signal Notification 2 Register, 107
SPU signal notification channels, 305
SPU signal notification facility, 105
SPU signal notification registers
SPU Signal Notification 1, 106
SPU Signal Notification 2, 107
SPU signalling channels, 142
SPU state management channels, 306
SPU Status Register, 97
SPU stop-and-signal instruction trap, 269
SPU Version Register, 289
SPU Write Decrementer Channel, 145
SPU Write Event Acknowledgment Channel, 161
SPU Write Event Mask Channel, 157
SPU Write Outbound Interrupt Mailbox Channel, 140
SPU Write Outbound Mailbox Channel, 139
SPU Write State Save-and-Restore Channel, 148
SPU_Cfg Register, 245
SPU_ChnlCnt Register, 244
SPU_ChnlData Register, 243
SPU_ChnlIndex Register, 242
SPU_ID Register, 291
SPU_In_Mbox Register, 103
SPU_LSLR, 241
SPU_NPC Register, 99
SPU_Out_Intr_Mbox, 237

SPU_Out_Mbox Register, 102
SPU_PrivCntl Register, 239
SPU_RdDec Register, 146
SPU_RdEventMask Channel, 159
SPU_RdEventStat Channel, 153
SPU_RdInMbox Channel, 141
SPU_RdMachStat Channel, 147
SPU_RdSigNotify1 Channel, 143
SPU_RdSigNotify2 Channel, 144
SPU_RdSRR0 Channel, 148
SPU_RunCntl Register, 96
SPU_Sig_Notify_1 Register, 106
SPU_Sig_Notify_2 Register, 107
SPU_Status Register, 97
SPU_VR Register, 289
SPU_WrDec Channel, 145
SPU_WrEventAck Channel, 161
SPU_WrEventMask Channel, 157
SPU_WrOutIntrMbox Channel, 140
SPU_WrOutMbox Channel, 139
SPU_WrSRR0 Channel, 148
stall-and-notify bit, for list elements, 63
state lost and isolated (SLI), power management, 285
state management channels, SPU, 306
State Register One, MFC, 221
state retained and isolated (SRI), power management, 285
states
  power management, 285
  SPU operating, 185
status channels
  MFC Read Atomic Command Status, 137
  MFC Read List Stall-and-Notify Tag Status, 135
  MFC Read Tag-Group Status, 133
  MFC Tag-Group Status, 126
  SPU Read Event Status, 153
  SPU Read Machine Status, 147
status of tag groups, determining, 126
Status Register, SPU, 97
status registers
  SPU Mailbox Status, 104
  SPU Status, 97
stdcx. instruction, 70
storage access ordering, 175
storage control attributes
  caching inhibited, 177
  guarded, 44, 177
  memory coherence required, 44, 75–76, 178
  overview, 44
  real-mode, 218
  write through required, 44
storage control commands, 67
storage control, real-mode facilities, 217
Storage Description Register, MFC, 224
storage domains
  illustration, 176
storage models
  description, 41
  local storage, 41
  virtual storage, 41
storage types, 30
stwcx. instruction, 70
suspend (Sc) bit, 233
symbols used, 17
sync instruction, 74
synchronization commands
  barrier, 60, 70–72, 76
  mfceieio, 60, 70–73
  mfcsync, 60, 70–73, 75, 314
  sndsig, 60
  sndsigb, 60
  sndsigf, 60
  table of, 314
synergistic processor unit (SPU)
  CBEA organization, 27
  description, 29
  overview, 49
system caused interrupts, 261

T

tag groups
  determining status, 126
  MFC Read Tag-Group Query Mask Channel, 131
  MFC Read Tag-Group Status Channel, 133
  MFC Write Tag Status Update Request Channel, 132
  MFC Write Tag-Group Query Mask Channel, 129
tag specific barrier command, 63
tag status channels, MFC, 306
tag-group completion condition, 93
tag-group completion interrupt, 270
tag-group status completion, example, 323
tag-independent barrier command, example, 322
tag-specific barrier, 55, 63
tag-specific fence, 55
TG (command tag parameter), 62, 311
TLB (translation lookaside buffer) management SPRs, 310
TLB Index Hint Register, 209
TLB Index Register, 210
TLB instructions
  tlbia, 216
  tlbie, 208, 214
  tlbiel, 214
TLB Invalidate All Register, 216
TLB Invalidate Entry Register, 214
TLB load control bit, 315
TLB mapping, 208
TLB Real Page Number Register, 212
TLB register names
  TLB_Index, 210

TLB_Index_Hint, 209
TLB_Invalidate_All, 216
TLB_Invalidate_Entry, 214
TLB_RPN, 212
TLB_VPN, 211
TLB registers
TLB Index, 210
TLB Index Hint, 209
TLB Invalidate All, 216
TLB Invalidate Entry, 214
TLB Real Page Number, 212
TLB Virtual Page Number, 211
TLB Virtual Page Number Register, 211
TLB_Index Register, 210
TLB_Invalidate_Entry Register, 214
TLB_RPN Register, 212
tlbie instruction, 208
TLBs, software management of, 315
trademarks, 2
Transfer Size or List Size Channel, MFC, 120
transfer size, sndsig command, 77
translation lookaside buffer (TLB) management SPRs, 310
translations, of addresses, 53
TS (transfer size parameter), 62, 311

U

UME (user mode environment), 35
user mode environment (UME), 35

V

vector/SIMD multimedia extension, 29, 48, 319
version control, 287
Version Register, MFC, 290
Version Register, SPU, 289
version registers
BP_VR, 287
CBEA-Compliant Processor Version Register, 287
MFC Version Register, 290
MFC_VR, 290
PPE Processor Version Register, 288
PVR, 288
SPU Identification Register, 291
SPU Version Register, 289
SPU_ID, 291
SPU_VR, 289
version SPRs, 309
virtual addresses
size of virtual address space, 31
virtual storage model, 41
virtual page, 41
Virtual Page Number Register, TLB, 211
Virtual Segment ID Register, SLB, 203
virtual storage model, 41
VMX, *see* vector/SIMD multimedia extension

W

write channels
MFC Write List Stall-and-Notify Tag Acknowledgment, 136
MFC Write Multisource Synchronization Request, 149
MFC Write Tag Status Update Request, 132
MFC Write Tag-Group Query Mask, 129
SPU Write Decrementer, 145
SPU Write Event Acknowledgment, 161
SPU Write Event Mask, 157
SPU Write Outbound Interrupt Mailbox, 140
SPU Write Outbound Mailbox, 139
SPU Write State Save-and-Restore, 148
Write Decrementer Channel, SPU, 145
Write Event Acknowledgment Channel, SPU, 161
Write Event Mask Channel, SPU, 157
Write List Stall-and-Notify Tag Acknowledgment Channel, MFC, 136
Write Multisource Synchronization Request Channel, MFC, 149
Write Outbound Interrupt Mailbox Channel, SPU, 140
Write Outbound Mailbox Channel, SPU, 139
Write State Save-and-Restore Channel, SPU, 148
Write Tag Status Update Request Channel, MFC, 132
Write Tag-Group Query Mask Channel, MFC, 129
write through required, storage control attribute, 44

Z

zeroing reserved registers, 33