IBM PowerPC 970MP RISC Microprocessor User's Manual

Version 2.1

July 31, 2006

Printed in the United States of America July 2006

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 Microelectronics home page can be found at **ibm.com**/chips

July 31, 2006

Contents

List of Figures **15**

List of Tables **17**

Revision Log **21**

About This Book **23**

Audience 23
Organization 23
Related Documents 24
Companion Manuals 24
Additional Documentation 25
General PowerPC Documentation 26
Conventions 27
Acronyms and Abbreviations 28
Terminology Conventions 33

1. PowerPC 970MP Overview **35**

1.1 PowerPC 970MP Microprocessor Overview 35
1.2 PowerPC 970MP Functional Units 38
1.2.1 Introduction 38
1.2.1.1 Key Design Fundamentals of the Microprocessor Core 38
1.2.1.2 Detailed Features of the Microprocessor Core 39
1.3 970MP Dual-Core Module 43

2. Programming Model **45**

2.1 970MP Processor Register Set 45
2.1.1 Architected Registers in the 970MP Implementation 51
2.1.1.1 MSR Register (MSR) 51
2.1.1.2 Machine Status Save/Restore Register (SRR1) 51
2.1.1.3 Time Base and Decrementer (TB, DEC) 52
2.1.1.4 Processor ID Register (PIR) 52
2.1.2 PowerPC 970MP-Specific Registers 52
2.1.2.1 Move To and Move From System Register Instructions 52
2.1.2.2 HID Registers (HID0, HID1, HID4, and HID5) 56
2.1.2.3 Data Address Breakpoint Register (DABR) 63
2.1.2.4 Instruction Address Breakpoint Register (IABR) 64
2.1.2.5 Instruction Match CAM Array Access Register (IMC) 64
2.1.2.6 Performance Monitor Registers (MMCR0, MMCR1, MMCRA, PMC1-8) 65
2.1.2.7 Sampled Instruction Address and Sampled Data Address Registers (SIAR, SDAR) 66
2.1.2.8 Scan Communication Registers (SCOMC and SCOMD) 66
2.1.2.9 Hypervisor Decrementer Interrupt Register (HDEC) 67
2.1.2.10 Hypervisor Save/Restore Register (HSRR0, HSRR1) 67
2.1.2.11 Hypervisor SPRGs (HSPRG0, HSPRG1) 67
2.1.2.12 Trigger Registers (TRIG0, TRIG1, TRIG2) 67

2.1.2.13 Hardware Interrupt Offset Register (HIOR) 68
2.2 Instruction Set Summary 69
2.2.1 Classes of Instructions 69
2.2.1.1 Definition of Boundedly Undefined 70
2.2.1.2 Defined Instructions 70
2.2.1.3 Invalid Forms 71
2.2.1.4 Illegal Instructions 71
2.2.1.5 Reserved Instructions 72
2.2.2 Instruction Set Overview 72
2.2.3 Fixed-Point Processor 72
2.2.3.1 Fixed-Point Arithmetic and Compare Instructions 72
2.2.3.2 Fixed-Point Logical, Rotate, and Shift Instructions 73
2.2.3.3 Move to and Move from System Register Instructions 73
2.2.3.4 Move to and Move from Machine State Register 73
2.2.3.5 Fixed-Point Invalid Forms and Undefined Conditions 74
2.2.4 Floating-Point Processor 75
2.2.4.1 Floating-Point Arithmetic Instructions 75
2.2.4.2 Floating-Point Invalid Forms and Undefined Conditions 75
2.2.5 Vector Processor 75
2.2.6 Load Store Processor 76
2.2.6.1 Floating-Point Load-and-Store Instructions 76
2.2.6.2 Fixed-Point Load Instructions 76
2.2.6.3 Fixed-Point Store Instructions 76
2.2.6.4 Fixed-Point Load-and-Store Multiple Instructions 76
2.2.6.5 Fixed-Point Load-and-Store String Instructions 77
2.2.6.6 Load/Store Invalid Forms and Undefined Conditions 78
2.2.7 Branch Processor 79
2.2.7.1 Branch Processor Instructions 79
2.2.7.2 Branch Processor Instructions with Undefined Results 79
2.2.7.3 Move To Condition Register Fields Instruction 80
2.2.8 Storage Control Instructions 80
2.2.8.1 Key Aspects of Storage Control Instructions 80
2.2.8.2 Instruction Cache Block Invalidate (**icbi**) 81
2.2.8.3 Instruction Cache Synchronize (**isync**) 81
2.2.8.4 Data Cache Block Touch (**dcbt** and **dcbtst**) 81
2.2.8.5 Data Cache Block Zero (**dcbz**) 82
2.2.8.6 Data Cache Block Store (**dcbst**) 82
2.2.8.7 Data Cache Block Flush (**dcbf**) 83
2.2.8.8 Load and Reserve and Store Conditional Instructions (**lwarx/ldarx**, **stwcx/stdcx**) 83
2.2.9 Memory Synchronization Instructions 83
2.2.10 Recommended Simplified Mnemonics 84

3. Storage Subsystem 85
3.1 Storage Hierarchy 85
3.2 Caches 86
3.2.1 Store Gathering 86
3.3 Storage Model 87
3.3.1 Atomicity 87
3.3.2 Storage Access Ordering 87
3.3.2.1 Storage Access Alignment Support 87

3.3.3 Atomic Updates and Reservations 88
3.4 Cache Management 89
3.4.1 Flushing the L1 I-Cache 89
3.4.2 Flushing the L1 D-Cache 89
3.4.3 L2 Cache Disabling and Enabling 89
3.4.4 L2 Cache Flushing 89
3.4.4.1 L2 Cache Flush in Direct-Mapped Mode 89
3.4.5 L2 Cache Flush Algorithm 90
3.5 Functional Units 92
3.5.1 Core Interface Unit 93
3.5.2 L2 Cache Controller 93
3.5.2.1 Cache Coherency 95
3.5.2.2 Cache-Coherency Paradoxes 95
3.5.2.3 Cache State Transition Tables 95
3.5.3 Non-Cacheable Unit 98
3.5.3.1 NCU Queues 99
3.5.3.2 Downstream Operations 100
3.5.3.3 Upstream Operations (Snoops) 102
3.5.3.4 Transfer Type Mapping 102
3.5.3.5 Bus Interface Unit 108
3.5.4 Data Prefetch 110
3.5.4.1 Overview of the Hardware-Controlled Data Prefetch 110
3.5.4.2 Hardware Prefetch Engine Implementation 112
3.5.4.3 Managing the Data Prefetch Hardware 120
3.5.4.4 Vector Prefetch Instruction Support 123
3.5.4.5 Programmability 125

4. Exceptions 127
4.1 970MP Microprocessor Exceptions 128
4.2 Exception Recognition and Priorities 131
4.2.1 Exception Priorities 131
4.3 Exception Processing 133
4.3.1 Machine Status Save/Restore Register 0 (SRR0) 133
4.3.2 Machine Status Save/Restore Register 1 (SRR1) 133
4.3.3 Machine State Register (MSR) 134
4.3.4 Enabling and Disabling Exceptions 136
4.3.5 Exception Processing Steps 136
4.3.6 Setting the Recoverable Exception in the MSR 137
4.3.7 Returning from an Exception Handler 137
4.4 Process Switching 138
4.5 Exception Definitions 138
4.5.1 System Reset Exception 138
4.5.2 Machine Check Exceptions 139
4.5.3 Data Storage Exception 141
4.5.4 Data Segment Exception 141
4.5.5 Instruction Storage Exception 141
4.5.6 Instruction Segment Exception 141
4.5.7 External Interrupt Exception 142
4.5.8 Alignment Exception 142

4.5.9 Program Exception 142
4.5.10 Floating-Point Unavailable Exception 143
4.5.11 Decrementer Exception 143
4.5.12 System Call Exception 143
4.5.13 Trace Exception 143
4.5.14 Performance Monitor Exception 144
4.5.15 VPU Unavailable Exception 144
4.5.16 Instruction Address Breakpoint Exception 145
4.5.17 Maintenance Exception 145
4.5.18 VPU Assist Exception 145

5. Memory Management 147
5.1 MMU Overview 147
5.1.1 Speculative Storage Accesses 148
5.1.2 Storage Protection 149
5.1.3 Storage Access Modes 149
5.1.4 Support for 32-Bit Operating Systems 149
5.2 Real Addressing Mode 150
5.3 Memory Segment Model 150
5.3.1 Segment Table 150
5.3.1.1 Segment Lookaside Buffer 150
5.3.1.2 Address Space Register 151
5.3.2 Page Table 151
5.3.2.1 Translation Lookaside Buffer 151
5.3.2.2 Large Page Support 151
5.3.2.3 Reference and Change Bits 152
5.3.2.4 TLB Invalidate Entry Instructions 152
5.3.2.5 TLB Invalidate All Instruction 153
5.3.2.6 TLB Synchronize Instruction 153
5.3.3 Effective-to-Real-Address Translation Caches 154
5.3.3.1 Instruction Effective-to-Real-Address Translation Cache 154
5.3.3.2 Data Effective-to-Real-Address Translation Cache 154

6. Performance 157
6.1 Pipeline Structure of the Microprocessor Core 157
6.2 Branch Processor 159
6.2.1 Instruction Fetching 159
6.2.2 Branch Prediction 159
6.2.3 Special Branch and Link Instruction: BCL 20, 31, $+4 159
6.2.4 Instruction Cache Touch Hint 160
6.2.5 Out-of-Order Execution and Instruction Flushes 160
6.3 Performance Features, Mechanisms, and Hazards 160
6.3.1 L1 Instruction Cache and Prefetching 160
6.3.1.1 Instruction Prefetching 160
6.3.1.2 Predecode Bits 161
6.3.1.3 Instruction Fetch Alignment 162
6.3.1.4 L1 I-Cache Indexing and Replacement Algorithm 162
6.3.1.5 **ICBI** Instruction Handling 163
6.3.1.6 Instruction Fetch Address Translation 163

6.3.1.7 Instruction Fetching while in Real Mode (MSR[IR] equals '0') 164
6.3.2 Branch Prediction 164
6.3.2.1 Branch Direction Prediction Using the Branch History Tables 165
6.3.2.2 Branch Prediction Using Static Prediction and the "a" and "t" Bits 166
6.3.2.3 Address Prediction Using the Link Stack 167
6.3.2.4 Address Prediction using the Count Cache 168
6.3.2.5 Round-Trip Branch Processing 169
6.3.3 Instruction Decode, Cracking, and Microcode 170
6.3.3.1 In-line Dataflow Instruction Cracking 171
6.3.3.2 Microcode Template-Based Instruction Cracking 175
6.3.3.3 Run Time Feedback-Based Instruction Cracking 178
6.3.3.4 Unused Cycles in the Flow of Dispatch Groups 178
6.3.4 Instruction Dispatch, Grouping, and Interlocks 179
6.3.4.1 Resource-Based Instruction Grouping 179
6.3.4.2 Synchronization-Based Instruction Grouping 180
6.3.4.3 Instruction Grouping Based on Flush Templates 183
6.3.5 Register Renaming 183
6.3.5.1 GPR Renaming 184
6.3.5.2 Link and Count Register Renaming 184
6.3.5.3 Condition Register Renaming 184
6.3.5.4 FPR Renaming 184
6.3.5.5 XER Renaming 185
6.3.5.6 FPSCR Renaming 185
6.3.5.7 VRF, VRSAVE, and VSCR Renaming 185
6.3.6 Instruction Issue and Register File Access 186
6.3.7 L1 Data Cache Management 187
6.3.7.1 L1 D-Cache Indexing and Replacement Algorithm 187
6.3.7.2 Misaligned Storage References 187
6.3.8 Load Instruction Handling 187
6.3.8.1 Cache Misses on Loads 187
6.3.8.2 Load Reorder Queue 188
6.3.8.3 Data Address Translation 188
6.3.8.4 Loads in Real Mode 189
6.3.8.5 Round-Trip Load Processing 189
6.3.9 Store Instruction Handling 193
6.3.9.1 Store Queue and Store Forwarding 193
6.3.9.2 Cache Misses on Stores 193
6.3.9.3 Store Gathering 193
6.3.9.4 Stores in Real Mode 193
6.3.9.5 Round-Trip Store Processing 194
6.3.10 Storage Access Delays 196
6.3.10.1 Load/Store Reject Mechanism 196
6.3.10.2 Flushing Storage Access Conflicts 198
6.3.11 Handling Other Core Instructions 199
6.3.11.1 Condition Register Logical Instructions 199
6.3.11.2 Synchronization Instructions 200
6.3.11.3 Cache and Synchronization Operations in STS 201
6.3.11.4 System Linkage Instructions 201
6.3.11.5 Stalls in Floating-Point Instruction Processing 201
6.3.11.6 Delays in Instruction Processing 202

6.4 Key Latencies, Throughputs, and Bandwidths 203
6.4.1 Instruction Latencies and Throughputs 203
6.4.1.1 Branch Instruction Characteristics 204
6.4.1.2 Instruction Characteristics (Condition Register Logic) 204
6.4.1.3 Instruction Characteristics (Load/Store) 205
6.4.1.4 Instruction Characteristics (FXU) 208
6.4.1.5 Instruction Characteristics (FPU) 211
6.4.1.6 Instruction Characteristics (Miscellaneous) 212
6.4.1.7 Instruction Characteristics (Vector) 213
6.4.2 Storage Alignment Performance Characteristics 215
6.4.3 Memory System Performance Characteristics 217
6.4.3.1 Load Latencies 217
6.4.3.2 Peak System Bandwidths 217

7. Signal Description 219
7.1 Signal Configuration 220
7.2 Signal Descriptions 221
7.2.1 Processor Interface 221
7.2.1.1 Address/Data In (ADIN[0:43])–Input 221
7.2.1.2 Snoop Response In (SRIN[0:1], $\overline{\text{SRIN}}$[0:1])–Input 222
7.2.1.3 Clock In (CLKIN/$\overline{\text{CLKIN}}$)–Input 222
7.2.1.4 Address Data Out (ADOUT[0:43])–Output 223
7.2.1.5 Snoop Response Out (SROUT[0:1], $\overline{\text{SROUT}}$[0:1])–Output 223
7.2.1.6 Clock Out (CLKOUT/$\overline{\text{CLKOUT}}$)–Output 223
7.2.2 Processor Status and Control 223
7.2.2.1 Quiescent Request ($\overline{\text{CP0\_QREQ}}$ and $\overline{\text{CP1\_QREQ}}$)–Output 223
7.2.2.2 Quiescent Acknowledgment ($\overline{\text{CP0\_QACK}}$ and $\overline{\text{CP1\_QACK}}$)–Input 224
7.2.2.3 Time-Base Enable (TBEN)–Input 224
7.2.2.4 Processor ID (PROCID[0:1])–Input 224
7.2.2.5 Bus Configuration Select (BUSCFG[0:2])–Input 224
7.2.2.6 PLL Locked (PLL_LOCK)–Output 225
7.2.2.7 Clock Receiver Termination (CKTERM_DIS)–Input 225
7.2.3 Clock Control 225
7.2.3.1 System Clock (SYSCLK/$\overline{\text{SYSCLK}}$)–Input 225
7.2.3.2 Phase Synchronization (psync)–Input 225
7.2.3.3 PLL Bypass ($\overline{\text{BYPASS}}$)–Input 225
7.2.3.4 PLL Multiplier (PLL_MULT)–Input 226
7.2.3.5 PLL Range Select (PLL_RANGE[0:1])–Input 226
7.2.4 Interrupts and Resets 226
7.2.4.1 Interrupt ($\overline{\text{CP0\_INT}}$ and $\overline{\text{CP1\_INT}}$)–Input 226
7.2.4.2 Machine Check Interrupt ($\overline{\text{MCP}}$)–Input 226
7.2.4.3 Checkstop ($\overline{\text{CHKSTOP}}$) –Bidirectional 226
7.2.4.4 Hard Reset ($\overline{\text{CP0\_HRESET}}$ and $\overline{\text{CP1\_HRESET}}$)–Input 227
7.2.4.5 Soft Reset ($\overline{\text{CP0\_SRESET}}$ and $\overline{\text{CP1\_SRESET}}$)–Input 227
7.2.5 Debug/Test Interface 227
7.2.5.1 Attention (ATTENTION)–Output 227
7.2.5.2 Processor Interface Disable (EI_DISABLE)–Input 227
7.2.5.3 Trigger Out (TRIGGEROUT)–Output 227
7.2.5.4 JTAG Signals 227
7.2.5.5 I^2C Signals 228

7.2.6 Voltage and Ground 228

8. Processor Interconnect Bus 229

8.1 Overview 230
8.1.1 Packets 231
8.1.2 Bus Segments 231
8.1.2.1 Address/Data Bus Segment 231
8.1.2.2 Transfer-Handshake Bus Segment 232
8.1.2.3 Snoop-Response Bus Segment 232
8.1.3 Transactions 232
8.1.3.1 Read Transaction 233
8.1.3.2 Write Transaction 234
8.1.3.3 Command-Only Transaction 235
8.1.4 Memory and Cache Coherency 236
8.1.4.1 Physical Memory Size 236
8.1.4.2 Coherency Protocol 236
8.1.4.3 Coherency Block Size 236
8.2 Packet Transfer Protocol 237
8.2.1 Command Packet Definition 237
8.2.1.1 Address Modifiers 237
8.2.1.2 Transfer Type Field 239
8.2.1.3 Tag Definition 241
8.2.1.4 Command Pacing 241
8.2.2 Data Packet Definition 242
8.2.2.1 Two-Beat Transfers 243
8.2.2.2 Multi-Beat Transfers 243
8.2.3 Transfer-Handshake Packets 245
8.2.3.1 Null Transfer Handshake 246
8.2.3.2 Transfer-Handshake Acknowledgment 246
8.2.3.3 Transfer-Handshake Retry 247
8.2.3.4 Transfer-Handshake Parity Error 248
8.3 Snoop Responses 248
8.3.1 Snoop-Response Bus Implementation 249
8.3.2 Snoop-Response Descriptions 250
8.3.2.1 SResp Retry Response Code (Priority 1 - highest) 250
8.3.2.2 SResp Modified-Intervention Response Code (Priority 2) 251
8.3.2.3 SResp Shared-Intervention Response Code (Priority 3) 251
8.3.2.4 SResp Modified Response Code (Priority 4) 252
8.3.2.5 SResp Shared Response Code (Priority 5) 252
8.3.2.6 SResp Null or Clean Response Code (Priority 6 - lowest) 252
8.4 Bus Transactions 253
8.4.1 Terms 253
8.4.2 Memory Read Transactions (General) 254
8.4.2.1 Read Transaction 254
8.4.2.2 Read with No Intent to Cache Transaction 255
8.4.2.3 Read with Intent to Modify Burst Transaction 256
8.4.2.4 LARX-Reserve Transaction 256
8.4.3 Memory Write Transactions (General) 257
8.4.3.1 Write-With-Kill Transaction 257
8.4.3.2 Write-With-Clean Transaction 258

8.4.3.3 Write-With-Flush Transaction 258
8.4.4 Command-Only Transactions 258
8.4.4.1 DCLAIM Transaction (Invalidate Others) 258
8.4.4.2 Flush Transaction 259
8.4.4.3 Clean Transaction 259
8.4.4.4 IKill Transaction 259
8.4.4.5 TLBIE Transaction 260
8.4.4.6 TLBSYNC Transaction 260
8.4.4.7 SYNC Transaction 260
8.4.4.8 EIEIO Transaction 261
8.4.4.9 Null Transaction 261

9. Power and Thermal Management 263
9.1 Definitions 263
9.1.1 Full Power Mode 263
9.1.2 Doze Mode 263
9.1.3 Nap Mode 263
9.1.4 Deep Nap Mode 264
9.1.5 Dynamic Power Management 264
9.2 Power-Management Support 264
9.2.1 Power-Management Control Bits 264
9.2.2 Interrupts 265
9.2.3 Bus Snooping 265
9.2.3.1 Delay Calculation 268
9.2.4 Thermal Diodes 269
9.2.5 Bus States while in Power Saving Modes 269
9.3 Software Considerations for Power Management 270
9.3.1 Entering Power Saving Mode 270
9.3.2 External Interrupt Enable 270
9.4 Power Tuning Facility Overview 271
9.4.1 Power Tuning Facility Definitions 271
9.4.2 Power Modes 273
9.4.3 Power Transition Latencies 276
9.4.3.1 Idle to Run Transitions 277
9.4.3.2 Exiting Deep Nap Using a Decrementer Interrupt 278
9.4.3.3 Frequency Transitions in the Power Tuning Facility 278
9.5 PLL Design 279
9.6 Time-Base and Decrementer Registers 281
9.7 I^2C Bus Interface 281
9.8 Frequency and Voltage Scaling 281
9.8.1 Frequency Scaling 281
9.8.1.1 Initiating a Frequency Change 281
9.8.1.2 Power Control Register 284
9.8.1.3 Power Control Register High (PCRH) 286
9.8.1.4 Power Status Register 287
9.8.2 Power Adjustment Bus Transaction 288
9.8.3 Clock Dithering 290
9.8.4 Voltage Scaling 291
9.8.5 Frequency and Voltage Scaling Latencies 292

9.9 Reducing Clock Mesh Power 293
9.9.1 Power Saving in Deep Nap 293
9.10 Additional Dynamic Power Management 294

10. 970MP Performance Monitor 295

10.1 Instrumentation Facilities Overview 295
10.1.1 Performance Monitor Facilities 296
10.1.2 Performance Monitor Event Selection 296
10.1.3 Machine States and Enabling the Performance Monitor Counters 296
10.1.4 Trigger Events and Enabling the Performance Monitor Counters 296
10.1.5 Performance Monitor Exceptions 296
10.1.6 Sampling 297
10.1.7 Thresholding 297
10.1.8 Trace Support Facilities 297
10.2 Instruction Sampling Facilities 297
10.2.1 Special Purpose Registers and Fields Associated with Instrumentation 297
10.3 Performance Monitor Components 300
10.4 Performance Monitor Control Registers 301
10.4.1 Performance Monitor Control Register MMCR0 301
10.4.2 Performance Monitor Control Register MMCR1 304
10.4.3 Performance Monitor Control Register MMCRA 307
10.4.4 Performance Monitor Count Registers PMC1 - 8 309
10.4.5 Performance Monitor and Trace Related Bits in the Machine State Register (MSR) 310
10.4.6 Performance Monitor Related Bits in Hardware Implementation-Dependent Register 0 (HID0) 311
10.4.7 Performance Monitor Related Bits in the Control Register (CTRL) 311
10.4.8 Performance Monitor Related Bits in the SCOM0240, 1240 Register (SCOM x'240') 312
10.4.9 Performance Monitor Related Bits in the SCOM0360,1360 Register (SCOM x'360') 313
10.4.10 Performance Monitor Related Bits in the IMC Array (IMC) 314
10.4.11 Performance Monitor Related Bits in the Sampled Instruction Address Register (SIAR) 314
10.4.12 Performance Monitor Related Bits in the Sampled Data Address Register (SDAR) 314
10.4.13 Performance Monitor Related Bits in the SRR1 (SRR1) 315
10.4.14 Performance Monitor Related Bits in the Time-Base Register (TB) 316
10.5 Performance Monitor Event Selection 317
10.5.1 Direct Events 318
10.5.1.1 Combined Events 318
10.5.1.2 Source-Encoded Events 318
10.5.1.3 Instruction Counts 319
10.5.2 Over 32-Bit Count 322
10.5.2.1 Examples of Over Bit Count 322
10.5.3 Speculative Count 322
10.6 Configuring the Performance Monitor Bus 323
10.7 Enabling the Performance Monitor Counters 333
10.7.1 Machine States 333
10.7.2 Trigger Events 334
10.7.2.1 Time-Base Transition Events 335
10.7.2.2 PMC1 Counter Negative Condition Events 335
10.7.2.3 PMCj ($2 \leq j \leq 8$) Counter Negative Condition Events 336
10.7.3 Method for Enabling or Disabling Performance Monitor Counting 336

10.8 Performance Monitor Exceptions 337
10.9 Instruction Matching and Sampling 338
10.9.1 Stage 1 Eligibility 338
10.9.2 Stage 2 Eligibility 338
10.9.3 Stage 3 Eligibility 338
10.10 IFU Instruction Matching Facility 339
10.10.1 Overview of the IFU Instruction Matching Facility 339
10.10.2 IMC Array 340
10.10.3 Reading the IMC SPR with the **mfimc** Instruction 341
10.10.4 Writing the IMC SPR With the **mtimc** Instruction 342
10.10.5 The v0 and v1 Mask Criteria 343
10.10.6 Instruction Matching Examples 344
10.11 IDU Instruction Sampling Facility 344
10.11.1 Overview of the IDU Instruction Sampling Facility 344
10.11.2 Stage 1 Eligibility 345
10.11.3 Stage 2 Eligibility 347
10.11.4 Stage 3 Mark/No Mark 348
10.11.5 Complete Masking, Matching, and Marking Cycle 350
10.11.6 Examples of Instruction Sampling Scenarios 351
10.11.7 Enabling and Disabling Marking 354
10.12 SIAR and SDAR Registers 355
10.12.1 Instruction Sampling 355
10.12.1.1 Performance Monitor Exceptions 355
10.12.2 Single Step and Branch Trace Marking Mode 356
10.12.2.1 Single Step Trace Mode 356
10.12.2.2 Branch Trace Mode 357
10.12.3 Comparison to Previous PowerPC Processors 357
10.13 Thresholding 357
10.14 Detailed Event Information 360

11. System Design 369
11.1 I^2C Interface 369
11.2 Bus Initialization, Configuration, Power Management, and Test 369
11.2.1 Bus Initialization 369
11.2.2 Configurable Parameters 369
11.2.3 Configuration Interface 372
11.2.3.1 Processor Configurable Timing Delay Parameter Register (BUSCONF) 373
11.2.3.2 North Bridge Configurable Timing Delay Parameter Register 374
11.2.4 Power Management 375
11.2.5 Reliability, Availability, and Serviceability (RAS) Requirement 377
11.3 Processor Interconnect Electrical Interface 378
11.3.1 Initialization at Power-On Reset 379
11.3.2 Target Cycle 379
11.4 Processor Interconnect Bus Error Detection and Correction 381
11.4.1 Error Detection for Balanced Encoding 381
11.4.2 Error Detection for Alternative Encodings 381
11.4.2.1 Single-Error and Double-Error Detection 382
11.4.2.2 Single-Error Correct, Double-Error Detection 382

12. SCOM Interface and Registers 385
12.1 Processor Core SCOM SPR Access 385
12.1.1 Operating System Protocol to Access SCOM SPRs 385
12.1.2 SCOMD Format 386
12.1.3 SCOMC Format 386
12.2 SCOM Address Allocation 388
12.2.1 Register Description Conventions 392
12.2.2 SCOM Error Handling 392
12.2.3 Access Status Register 393
12.3 Core Pervasive SCOM Register Definitions 394
12.3.1 Processor CoreRAS Facilities (x'02[1:4]XXX') 394
12.3.2 Processor Core SPR SCOM Access (x'023XXX') 409
12.3.3 Processor Core Performance Monitor Sampling Control (x'02400X') 416
12.3.4 Processor Core FIR Facilities (x'03[0:5]XXX') 417
12.3.5 Instruction Mark Configuration (x'03600X') 424
12.4 Storage Subsystem SCOM Register Definition 426
12.4.1 L2 SCOM Register Definition 426
12.4.2 BIU SCOM Register Definition 429
12.4.3 Processor Interconnect Registers 436
12.5 Chip Pervasive SCOM Register Definition 446
12.5.1 Power-On Reset Registers (x'40XXXX') 446
12.5.2 Chip Free-Running Clock Section Control/Status (x'50[0:4]XXX') 456
12.5.3 Chip Parallel SCOM Control (x'6XXXXX') 464
12.5.4 Chip Clock/Scan Control (x'8[0:4]XXXXX') 470

13. Vector Processing Unit 491
13.1 970MP Vector and SIMD Multimedia Overview 491
13.1.1 VPU Implementation 491
13.1.2 Vector ALU 492
13.2 Vector Registers 493
13.2.1 VRSAVE Register 493
13.2.2 Vector Status and Control Register (VSCR) 493
13.3 Effects on Existing PowerPC Facilities 495
13.3.1 Control Flow 495
13.3.1.1 Condition Register 495
13.3.1.2 Machine State Register 496
13.3.1.3 Machine Status Save/Restore Registers (SRR0, SRR1) 497
13.4 Exceptions 498
13.4.1 VPU Unavailable Exception 498
13.4.2 VPU Assist Exception 498
13.4.3 Data Storage Exception 498
13.5 Optional Instructions 498
13.5.1 Java Mode Instruction Handling Implementation 498
13.5.2 Least Recently Used Instructions 499
13.5.3 Data Stream Instructions 499
13.6 Vector Instruction Set 499

List of Figures

Figure 1-1. 970MP Block Diagram 36
Figure 1-2. 970MP Dual Core with Common Arbitration Logic 37
Figure 2-1. 970MP Programming Model—Registers 46
Figure 2-2. Processor Attention Instruction 79
Figure 3-1. 970MP Storage Subsystem 92
Figure 3-2. Data Flow in the 1-MB L2 Cache 94
Figure 3-3. NCU Block Diagram 98
Figure 3-4. NCU Master Store and Load Queues 99
Figure 3-5. NCU Master Barrier Operations Flow 104
Figure 3-6. NCU Master LWSYNC Operations Flow 105
Figure 3-7. NCU Master EIEIO Operations Flow 106
Figure 3-8. BIU Address and Data Flow 109
Figure 3-9. Prefetch Engine Block Diagram 112
Figure 3-10. Prefetch Filter Queue Logic 115
Figure 3-11. Prefetch Request Queue (PRQ) 119
Figure 3-12. Data Cache Block Touch X-Form (Optional Variant) 120
Figure 3-13. Data Cache Block Touch X-Form (Enhanced Variant) 121
Figure 5-1. TLB Invalidate Entry Local X-Form (Processor Local Form) 153
Figure 6-1. Microprocessor Core Pipeline Structure 158
Figure 7-1. 970MP Microprocessor Signal Groups 220
Figure 7-2. Encoding and Selection Logic for the Drive Side of a 970MP Interconnect SSB 222
Figure 8-1. Processor Interconnect Bus Configuration with Two 970MP Microprocessors 229
Figure 8-2. Two Microprocessors Connected to a North Bridge 230
Figure 8-3. Read Transaction Timing Diagram 233
Figure 8-4. Write Transaction Timing Diagram 234
Figure 8-5. Command-Only Transaction Timing Diagram 235
Figure 9-1. Processor QREQ/QACK Signalling 266
Figure 9-2. North Bridge QREQ/QACK Signalling 267
Figure 9-3. Using a 970MP Microprocessor with a Single QREQ/QACK Pair 268
Figure 9-4. 970MP Power Mode States 273
Figure 9-5. PLL Design 280
Figure 9-6. Frequency Scaling Event Ordering 283
Figure 9-7. Clock Dithering Block Diagram 290
Figure 9-8. Sample Shift Pattern 291
Figure 10-1. Performance Monitor Architecture 300
Figure 10-2. Event Selection 317
Figure 10-3. 970MP Performance Monitor Bus Configuration 324
Figure 10-4. Patch Map 341

Figure 10-5. IFU and IDU Instruction Sampling Flow 349
Figure 10-6. Performance Monitor Threshold Logic 358
Figure 11-1. Configurable Timing Parameters 370
Figure 11-2. North Bridge Configurable Timing Parameters 370
Figure 11-3. Processor Configurable Timing Parameters 371
Figure 11-4. Processor QREQ and QACK Signalling 376
Figure 11-5. North Bridge QREQ and QACK Signalling 377
Figure 11-6. Bus Diagram of a Dual-Processor 970MP Processor Interconnect-Based System 378
Figure 11-7. Receive-Side FIFO Circuit 380
Figure 11-8. Timing Diagram Showing Relationship Between Bclk and the Four Gate Signals 380
Figure 12-1. Processor Unit SCOM Topology 385
Figure 12-2. SCOMC SPR Format 386
Figure 12-3. Format of an SCOM Address 388
Figure 12-4. Format of an SCOM Address within the BIU 388
Figure 12-5. Format of an SCOM Data Bus 388
Figure 12-6. Common Clock Commands 471
Figure 12-7. Example of LBIST Commands using the EPS Engine 479
Figure 13-1. VPU Block Diagram 492
Figure 13-2. VSCR Format 493
Figure 13-3. VSCR Moved to a Vector Register 494
Figure 13-4. Condition Register (CR) 495

List of Tables

Table i. Acronyms and Abbreviated Terms 28
Table ii. Terminology Conventions 33
Table iii. Instruction Field Conventions 33
Table 2-1. MSR Bits 51
Table 2-2. Additional SRR1 Bit 51
Table 2-3. Implementation-Specific SPRs 53
Table 2-4. Move To/Move From SPR Behavior 55
Table 2-5. Invalid Forms of Instructions 71
Table 2-6. Storage Control Instructions 80
Table 2-7. **dcbz** Actions 82
Table 3-1. Storage Hierarchy Characteristics 85
Table 3-2. Simple Address Decode 89
Table 3-3. Storage Subsystem Functional Units 92
Table 3-4. Cache-Coherency Protocol 95
Table 3-5. 970MP L2 Cache State Transitions Due to Processor Instructions 95
Table 3-6. 970MP L2 Cache State Transitions Due to Bus Operations 96
Table 3-7. CIU to Processor Interconnect Bus Transfer Type Mapping (Store Port) 102
Table 3-8. CIU to Processor Interconnect Bus Transfer Type Mapping (Load Port) 103
Table 3-9. Ramp-Up for Hardware-Initiated Stream 116
Table 3-10. Ramp-Up for Software-Initiated Stream 116
Table 3-11. Shift for N_S Based on Stride 124
Table 3-12. DST Mapping Summary 124
Table 3-13. DST Mapping Examples 125
Table 4-1. 970MP Microprocessor Exception Classifications 128
Table 4-2. Exceptions and Conditions 129
Table 4-3. IEEE Floating-Point Exception Mode Bits 136
Table 4-4. Register Settings for Machine Check Exception 140
Table 4-5. Register Settings for Alignment Exception 142
Table 4-6. Register Settings for Trace Exception 143
Table 4-7. Register Settings for the Performance Monitor Exception 144
Table 4-8. Register Settings for VPU Unavailable Interrupt 144
Table 4-9. Register Settings for Maintenance Exception 145
Table 4-10. Register Settings for VPU Assist Exception 145
Table 5-1. MMU Feature Summary 148
Table 5-2. Treatment of WIMG Bits in the 970MP 149
Table 6-1. Predecode Bits 161
Table 6-2. Handling **bclr** and **bclrl** Instructions 168
Table 6-3. Handling **bcctr** and **bcctrl** Instructions 169

Table 6-4. Pipeline Bubbles Due to Microcode 171
Table 6-5. Cracked Instructions 173
Table 6-6. Instructions with Group Formation Restrictions 174
Table 6-7. Microcoded Instructions (No Alignment) 175
Table 6-8. Microcoded Instruction (Misaligned Loads and Stores) 177
Table 6-9. Decode Slot/Dispatch Restrictions 179
Table 6-10. Instructions with Synchronizing Properties 182
Table 6-11. Conditions That Cause Instruction Rejection 196
Table 6-12. Conditions That Cause Flushing 198
Table 6-13. Issue-to-Issue Delays for Dependent Operations 202
Table 6-14. Branch Instruction Characteristics 204
Table 6-15. Instruction Characteristics (Condition Register Logic) 204
Table 6-16. Instruction Characteristics (Load/Store) 205
Table 6-17. Instruction Characteristics (FXU) 208
Table 6-18. Instruction Characteristics (FPU) 211
Table 6-19. Instruction Characteristics (Miscellaneous) 212
Table 6-20. Instruction Characteristics (Vector) 213
Table 6-21. Storage Alignment Characteristics 215
Table 6-22. Key Load Latencies 217
Table 6-23. Peak System Bandwidths 217
Table 8-1. Processor Interconnect Signal Description 231
Table 8-2. Command Packet Description 237
Table 8-3. Transfer Type Encoding 239
Table 8-4. Transfer Size Encoding 240
Table 8-5. Tag Definition 241
Table 8-6. Read-Data Packet Header Description 242
Table 8-7. Data Beat Description 242
Table 8-8. Two-Beat Data Transfers 243
Table 8-9. Packet Ordering for 128-Byte Interleaved Packets on 32-Byte Boundaries 244
Table 8-10. Packet Ordering for 32-Byte Interleaved Packets 244
Table 8-11. Transfer-Handshake Definition 245
Table 8-12. Snoop-Response Bit Definition 248
Table 8-13. Allowed Snoop Responses 249
Table 8-14. Write-With-Kill Types Supported 257
Table 9-1. Power-Management Control Bits 264
Table 9-2. Minimum QAckIdleDelay requirement in bus clocks for 970MP 269
Table 9-3. Minimum (QAckIdleDelay + QAckMinLowTime) requirement in bus clocks for 970MP 269
Table 9-4. Power-Management Modes 271
Table 9-5. Power Mode States 274

Table 9-6. Transitions between Power Modes 275
Table 9-7. Valid Combinations of Power Modes 276
Table 9-8. Latency of Deep-Nap-to-Run Transitions in Full Frequency Cycles 278
Table 9-9. Power Adjustment Transaction 288
Table 10-1. 970MP Performance Monitor and Trace-Related Special Purpose Registers 299
Table 10-2. Performance Monitor Internal Multiplexer PMCxSEL[0:4] Bit Values 317
Table 10-3. Event Data Source Encodings 318
Table 10-4. Event Instruction Source Encodings 319
Table 10-5. Direct Events 320
Table 10-6. Speculative Count Events 323
Table 10-7. Performance Monitor Bus Assignments 325
Table 10-8. Examples of Event Counter Enabling States 334
Table 10-9. Partial Match Rows in the IMC Array 340
Table 10-10. Complete Match Rows in the IMC Array 340
Table 10-11. IMC SPR Patch Map Sample Results 342
Table 10-12. IMC SPR for a 17-Bit Match 343
Table 10-13. IMC SPR Used when Writing the Second **mtimc** Instruction for a 32-Bit Match 343
Table 10-14. Encoding Bits v0 and v1 of the IMC Array Mask 344
Table 10-15. IFU BSFL Predecode Bit Definitions 346
Table 10-16. Start and End Event Select Bits and the Performance Monitor Threshold Logic 359
Table 10-17. Detailed Event Descriptions 360
Table 11-1. Programmable Delay Parameters 371
Table 11-2. I^2C Interface Signals 372
Table 11-3. I^2C Registers Used by the 970MP Processor Interconnect 372
Table 11-4. Bit Error Position Identifier 383
Table 12-1. Operating System Code to Access SCOM 386
Table 12-2. SCOMC SPR Format 386
Table 12-3. SCOM Base Addresses 389
Table 12-4. SCOM Modifier Addresses 389
Table 12-5. EPS Engine Description 478
Table 13-1. VSCR Field Descriptions 494
Table 13-2. CR6 Field Bit Settings for Vector Compare Instructions 495
Table 13-3. MSR Bit Settings Affecting the VPU 496
Table 13-4. Supported Vector Instructions 499

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 modified from the previous release of this document.

Revision Date	Page	Description
June 2, 2006	46 284, 286 385	Version 2.1 • Added SPRG3 to User Model — USIA block in *Figure 2-1 970MP Programming Model—Registers* and added a description. • Corrected the SCOM address of the Power Control Register (PCR). • Added commonly used SCOM registers and their descriptions (*Section 12 SCOM Interface and Registers*).
	268	• Added *Section 9.2.3.1 Delay Calculation*
June 28, 2005	52 78 243 263 264 371	Version 2.0 • Clarified the explanation of illegal instructions. • Rewrote the description of the Load with Update instructions. • Updated *Table 8-8 Two-Beat Data Transfers*, the description of the data transfer format, and *Table 8-9 Packet Ordering for 128-Byte Interleaved Packets on 32-Byte Boundaries*. • Expanded the description of Doze mode and how it relates to power management. • Corrected the programmable delay parameters for SNOOPLAT and SNOOPACC.
January 17, 2005		Version 1.0 (Initial release)

About This Book

The primary objective of the *IBM PowerPC 970MP RISC Microprocessor User's Manual* is to define the functionality of the PowerPC 970MP microprocessor for software and hardware developers.

The information in this book is subject to change without notice, as described in the disclaimers on the title page. As with any technical documentation, it is the readers' responsibility to be sure they are using the most recent version of the documentation. To locate any published errata or updates for this document, go to **ibm.com**/chips/techlib.

Note: Soft copies of many of the latest versions of the manuals and documents referred to in this manual that are produced by IBM can be accessed on the Web at **ibm.com**/chips/techlib.

Audience

This manual is intended for system software and hardware developers and application programmers who want to develop products for the 970MP microprocessor. It is assumed that the reader understands operating systems, microprocessor system design, basic principles of reduced instruction set computer (RISC) processing, and details of the PowerPC Architecture™.

Organization

For ease in reference, the arrangement of topics in this book is similar to that of the *PowerPC Microprocessor Family: The Programming Environments Manual for 64-Bit Microprocessors* and the *PowerPC Microprocessor Family: Vector/SIMD Multimedia Extension Technology Programming Environments Manual* (see *Related Documents* on page 24). Topics build upon one another, beginning with a description and summary of 970MP-specific registers and instructions and progressing to more specialized topics such as 970MP-specific details regarding the cache, exception, memory management models, and power management. Thus, chapters may include information from multiple levels of the architecture. For example, the discussion of the cache model uses information from both the virtual environment architecture (VEA) and the operating environment architecture (OEA).

A summary and a brief description of the major sections of this manual follows:

- *Chapter 1 PowerPC 970MP Overview* is useful for readers who want a general understanding of the features and functions of the PowerPC Architecture and the 970MP processor. This chapter describes the flexible nature of the PowerPC Architecture definition, and provides an overview of how the PowerPC Architecture defines the register set, operand conventions, addressing modes, instruction set, cache model, exception model, and memory management model.
- *Chapter 2 Programming Model* is useful for software engineers who need to understand the 970MP-specific registers, operand conventions, and details regarding how the PowerPC instructions are implemented on the 970MP microprocessor. Instructions are organized by function.
- *Chapter 3 Storage Subsystem* discusses the storage subsystem as implemented on the 970MP microprocessor. The storage subsystem includes the core interface logic, the non-cacheable unit, the L2 cache and controls, and the bus interface unit.
- *Chapter 4 Exceptions* describes the exception model defined in the PowerPC OEA and the specific exception model implemented on the 970MP microprocessor.

- *Chapter 5 Memory Management* describes the 970MP implementation of the memory management unit specifications provided by the PowerPC OEA for PowerPC processors.
- *Chapter 6 Performance* provides information about the performance profile of the 970MP microprocessor. Key performance related features and mechanisms, latencies, throughput, and bandwidth are discussed.
- *Chapter 7 Signal Description* describes the individual signals of the 970MP microprocessor.
- *Chapter 8 Processor Interconnect Bus* describes the processor interface (PI), which is a bus architecture providing high-speed, high-performance interconnections for processors, I/O devices, memory subsystems, and bridge chips.
- *Chapter 9 Power and Thermal Management* provides information about power saving and thermal management modes for the 970MP microprocessor.
- *Chapter 10 970MP Performance Monitor* describes the operation of the performance monitor diagnostic tool incorporated in the 970MP microprocessor and provides detailed event information.
- *Chapter 11 System Design* describes system-related features such as power-on reset and reliability, availability, and serviceability (RAS) considerations.
- *Chapter 13 Vector Processing Unit* provides a general understanding of the features and functions of the vector processing unit (VPU) used on the 970MP microprocessor.

Related Documents

Companion Manuals

This manual is intended as a companion to the following reference manuals:

- PowerPC Architecture[1] books:

  Note: The PowerPC Architecture books supersede the *PowerPC Programming Environments Manual* for the 970MP implementation. However, not all features available in the PowerPC Architecture are supported in the 970MP microprocessor (such as, logical partitioning).

  – *PowerPC User Set Architecture (Book I, Version 2.01)*. Covers the base user instruction set architecture (UISA), user-level registers, data types, memory conventions, memory and programming models, and related facilities available to the application programmer.

  – *PowerPC Virtual Environment Architecture (Book II, Version 2.01)*. Defines the storage model and related instructions and facilities available to the programmer, and the time-keeping facilities available to the application programmer. The VEA, which is the smallest component of the PowerPC Architecture, defines additional user-level functionality that falls outside typical user-level software requirements. The VEA describes the memory model for an environment in which multiple processors or other devices can access external memory and define aspects of the cache model and cache control instructions from a user-level perspective. The resources defined by the VEA are particularly useful for optimizing memory accesses and for managing resources in an environment in which other processors and other devices can access external memory.

    Implementations that conform to the PowerPC VEA also conform to the PowerPC UISA, but may not necessarily adhere to the operating environment architecture (OEA).

1. PowerPC Architecture refers to the instructions and facilities described in Books I, II, and III.

– *PowerPC Operating Environment Architecture (Book III, Version 2.01)*. Defines the system (privileged) instructions and related facilities. The OEA defines supervisor-level resources typically required by an operating system. The OEA defines the PowerPC memory management model, supervisor-level registers, and the exception model.

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

- *PowerPC Microprocessor Family: Programming Environments Manual for 64-Bit Microprocessors* (referred to as the *Programming Environments Manual*). Provides information about resources defined by the PowerPC Architecture that are common to PowerPC processors. This manual describes the functionality of the 64-bit architecture model.
- *PowerPC Microprocessor Family: Vector/SIMD Multimedia Extension Technology Programming Environments Manual*. Describes how the vector/SIMD technology relates to both the 64-bit and the 32-bit portions of the PowerPC Architecture.
- *The PowerPC Architecture: A Specification for a New Family of RISC Processors* by C. May, E. Silha, R. Simpson, and H. Warren, Morgan Kaufman, May 1994. Defines the architecture from the perspective of the three programming environments and remains the defining document for the PowerPC Architecture.

Because the PowerPC Architecture is designed to be flexible in order to support a broad range of processors, these documents provide a general description of features that are common to PowerPC processors and indicate those features that are optional or that may be implemented differently in the design of each processor.

It is important to note that some resources are defined more generally at one level in the architecture and more specifically at another. For example, conditions that cause a floating-point unavailable exception are defined by the UISA, while the exception mechanism itself is defined by the OEA.

Additional Documentation

Some additional PowerPC documentation is available at **ibm.com**/chips/techlib through IBM Customer Connect at http://ibm.com/technologyconnect.

- *IBM PowerPC 970MP RISC Microprocessor Datasheet.* This datasheet provides specific data about bus timing, signal behavior, and ac, dc, and thermal characteristics, as well as other design considerations for the 970MP implementation.
- *PowerPC 970MP Power On Reset Application Note*. This document contains information about required power-on-reset design and initialization.
- *PowerPC Microprocessor Family: The Programmer's Reference Guide* (MPRPPCPRG-01). This is a concise reference that includes the register summary, memory control model, exception vectors, and the PowerPC instruction set.
- Application notes. These short documents contain information about specific design issues useful to programmers and engineers working with PowerPC processors.

General PowerPC Documentation

The following documentation provides useful information about the PowerPC Architecture and computer architecture in general:

Ferraiolo, F., E. Cordero, D. Dreps, M. Floyd, "Power4: Synchronous Wave-Pipelined Interface." *Hot Chips 1999*, Stanford, CA.

Hennessy, John L. and David A. Patterson. *Computer Architecture: A Quantitative Approach*. 2nd ed.

I^2C-Bus Specification. Version 2.1. Philips Semiconductors, 2000.

IEEE Standard Test Access Port and Boundary-Scan Architecture, IEEE Std. 1149.1a-1993.

McClanahan, Kip*. PowerPC Programming for Intel Programmers.* Foster City, CA: IDG Books Worldwide, Inc.

Shanley, Tom. *PowerPC System Architecture*. Richardson, TX: Mindshare, Inc.

Conventions

This document uses the following notational conventions:

&	AND logical operator.
|	OR logical operator.
'0'	Binary values in text are either spelled out (zero and one) or appear in single quotation marks. For example: '10101'. In tables, these quotation marks are omitted.
¬	NOT logical operator.
crfD	Instruction syntax used to identify a destination CR field.
crfS	Instruction syntax used to identify a source Condition Register (CR) field.
frA, **fr**B, **fr**C	Instruction syntax used to identify a source Floating-Point Register (FPR).
frD	Instruction syntax used to identify a destination FPR.
italics	Italics indicate variable command parameters. For example, **bcctr***x*. Book titles in text are set in italics.
mnemonics	Instruction mnemonics are shown in lowercase bold.
n	Used to express an undefined numeric value.
$\overline{\text{overbar}}$	Overbars designate active-low (non-differential) signals.
rA, **r**B	Instruction syntax used to identify a source General-Purpose Register (GPR).
rD	Instruction syntax used to identify a destination GPR.
REG[FIELD]	Abbreviations or acronyms for registers are shown in uppercase text. Specific bits, fields, or ranges appear in brackets. For example, MSR[POW] refers to the Power-Management bit in the Machine State Register.
Reserved	Indicates reserved bits or bit fields in a register. Although these bits can be written to as either ones or zeros, they are always read as zeros.
vA, **v**B, **v**C	Instruction syntax used to identify a source Vector Register (VR).
vD	Instruction syntax used to identify a destination VR.
x	In certain contexts, such as a signal encoding, this indicates a don't care.
x'0'	A lowercase x precedes hexadecimal values. For example, x'0B00'.

Acronyms and Abbreviations

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

Table i. Acronyms and Abbreviated Terms (Page 1 of 5)

Term	Meaning
ALU	arithmetic logic unit
AS	application system
ASR	Address Space Register
BAT	block address translation
BCM	balanced coding method
BHT	branch history table
BIST	built-in self test
BIU	bus interface unit
BPU	branch processing unit
BSDL	boundary-scan description language
CAM	content-addressable memory
CDF	critical data forward
CIU	core interface unit
CMOS	complementary metal-oxide semiconductor
COP	common on-chip processor
CQ	completion queue
CR	Condition Register
CRA	custom register array
CTR	Count Register
DABR	Data Address Breakpoint Register
DAR	Data Address Register
D-cache	data cache
DCMP	data translation lookaside buffer (TLB) compare
DEC	Decrementer Register
DMISS	data TLB miss address
DPM	dynamic power management
DSI	data storage interrupt
DSISR	Data Storage Interrupt Status Register. Register used to determine the source of a DSI exception.
DTLB	data translation lookaside buffer
EA	effective address
EAR	External Access Register
ECC	error checking and correction
eCR	emulation CR
eFPR	emulation FPR

Table i. Acronyms and Abbreviated Terms (Page 2 of 5)

Term	Meaning
eGPR	emulation GPR
ERAT	effective-to-real-address translation
FIFO	first-in-first-out
FPECR	Floating-Point Exception Cause Register
FPR	Floating-Point Register
FPSCR	Floating-Point Status and Control Register
FPU	floating-point unit
GCT	global completion table
GPIO	general-purpose I/O pins
GPR	General-Purpose Register
HID*n*	Hardware Implementation-Dependent Register
I^2C	inter-integrated circuit
IABR	Instruction Address Breakpoint Register
IAP	initial alignment pattern
I-cache	instruction cache
IEEE®	Institute for Electrical and Electronics Engineers
IFU	instruction fetch unit
IMC	Instruction Match CAM Register
IQ	instruction queue
ISI	instruction storage interrupt
ISU	instruction sequencer unit
ITLB	instruction translation lookaside buffer
JTAG	Joint Test Action Group
L2	secondary cache (level 2 cache)
L2C	L2 cache controller
LHR	load-hit-reload. A load presented through a load/store port to the LMQ matches an existing entry that has already initiated a request to the L2.
LHS	load-hit-store. A load presented through a load/store port to the store reorder queue (SRQ) matches an existing entry. Store forwarding may be attempted. If the store contains all the data required by the load, store forwarding can occur. If the store does not contain all the data required by the load, store forwarding cannot occur and the load is rejected or flushed.
LIFO	last-in-first-out
LMQ	load miss queue. An 8-entry queue, which tracks loads that miss the L1 and are awaiting data from the 970MP storage subsystem (STS). Each entry can handle two loads associated with a cache line.
LR	Link Register
LRU	least recently used
LSb	least-significant bit
LSB	least-significant byte
LSU	load/store unit. The unit in the microprocessor that executes load-and-store instructions.

Table i. Acronyms and Abbreviated Terms (Page 3 of 5)

Term	Meaning
MERSI	modified/exclusive/recent/shared/invalid cache-coherency protocol
MMCR*n*	Monitor Mode Control Registers
MMU	memory management unit
MRU	most recently used
MSb	most-significant bit
MSB	most-significant byte
MSR	Machine State Register
NaN	not a number
NCU	non-cacheable unit
NIA	next instruction address
no-op	no operation
NSA	next sequential address
NTC	next to complete
OEA	operating environment architecture
PFQ	data prefetch filter queue. Filter queue of 12 entries, which can detect data streams for prefetching.
PI	processor interface
PID	processor identification tag
PLL	phase-locked loop
PMC*n*	Performance Monitor Counter Registers
POR	power-on reset
POWER™	Performance Optimized with Enhanced Reduced Instruction Set Computing (RISC) Architecture
PRQ	data prefetch request queue. A prefetch queue of eight streams, which will be prefetched.
PTE	page table entry
PTEG	page table entry group
PVR	Processor Version Register
RAS	reliability, availability, and serviceability
RAW	read-after-write
RCQ	read/claim queue
RISC	reduced instruction set computing
RLM	random logic macro
RMCI	real mode cache inhibited
RTL	register transfer language
RWITM	read with intent to modify
RWNITM	read with no intent to modify
SCOM	scan communications
SCOMC	SCOM control
SCOMD	SCOM data

Table i. Acronyms and Abbreviated Terms (Page 4 of 5)

Term	Meaning
SDA	sampled data address register
SDQ	store data queue
SDR1	Register that specifies the page table base address for virtual-to-physical address translation.
SHL	store-hit-load
SHR	store-hit-reload. A committed store ready to write to the L1 data cache (D-cache) line that matches an existing LMQ entry. The store is stalled until the reload is complete.
SIAR	Sampled Instruction Address Register
SIMD	single-instruction, multiple-data
SIMM	signed immediate value
SLB	segment lookaside buffer
SMP	symmetric multiprocessor
SPR	Special Purpose Register
SR*n*	Segment Register
SRQ	store reorder queue. A 32-entry queue that tracks all stores active in the LSU.
SRR0	Machine Status Save/Restore Register 0
SRR1	Machine Status Save/Restore Register 1
SSB	source-synchronous bus
STE	segment table entry
STQ	store queue
STS	970MP storage subsystem, which includes core interface logic, a noncacheable unit, the L2 cache and controls, and the bus interface unit.
TB	timebase facility
TBL	Timebase Lower Register
TBU	Timebase Upper Register
TLB	translation lookaside buffer
TType	transfer type
UIMM	unsigned immediate value
UISA	user instruction set architecture
UMMCR*n*	User Monitor Mode Control Registers
UPMC*n*	User Performance Monitor Counter Registers
USIA	User Sampled Instruction Address Register
VA	virtual address
VALU	vector unit arithmetic logic unit (ALU)
VEA	virtual environment architecture
VPERM	vector permute unit
VPU, vector units	vector processing units within the core.
VR	Vector Register

Table i. Acronyms and Abbreviated Terms (Page 5 of 5)

Term	Meaning
VRF	Vector Register file
VRSAVE	Vector Save/Restore Register
VSCR	Vector Status and Control Register
WAR	write-after-read
WAW	write-after-write
WIMG	write-through/caching-inhibited/memory-coherency enforced/guarded bits
XER	Integer Exception Register, used to indicate conditions such as carries and overflows for integer operations.

Terminology Conventions

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

Table ii. Terminology Conventions

Architecture Specification	Current Manual
Data storage interrupt (DSI)	DSI exception
Extended mnemonics	Simplified mnemonics
Instruction storage interrupt (ISI)	ISI exception
Interrupt	Exception
Privileged mode (or privileged state)	Supervisor-level privilege
Problem mode (or problem state)	User-level privilege
Real address	Physical address
Relocation	Translation
Storage (locations)	Memory
Storage (the act of)	Access
Store in	Write back
Store through	Write through
Swizzling	Double-word swap

Table iii describes instruction field notation used in this manual.

Table iii. Instruction Field Conventions

Architecture Specification	Equivalent to:
BA, BB, BT	**crb**A, **crb**B, **crb**D
BF, BFA	**crf**D, **crf**S
D	d
DS	ds
FLM	FM
FRA, FRB, FRC, FRT, FRS	**fr**A, **fr**B, **fr**C, **fr**D, **fr**S
FXM	CRM
RA, RB, RT, RS	**r**A, **r**B, **r**D, **r**S
SI	SIMM
U	IMM
UI	UIMM
VA, VB, VT, VS	**v**A, **v**B, **v**D, **v**S
VEC	Vector/SIMD multimedia extension technology

1. PowerPC 970MP Overview

The IBM PowerPC 970MP reduced instruction set computer (RISC) microprocessor is an implementation of the PowerPC Architecture. This chapter provides an overview of the features of the 970MP microprocessor and includes two block diagrams showing the major functional components.

Note: The 970MP microprocessor incorporates two complete microprocessors on a single chip, along with some common logic to connect these microprocessors to a system. The terms microprocessor, processor, and processing unit are used interchangeably to describe each of the two individual processors. The term core refers to the instruction fetch and execution logic, including the L1 cache, but excluding the storage subsystem, of each processor. The term PowerPC 970MP or 970MP refers to the single chip module comprising the two processing units and the common logic.

1.1 PowerPC 970MP Microprocessor Overview

The 970MP microprocessor is a dual core, 64-bit PowerPC RISC microprocessor with vector processing unit (VPU) extensions—the single-instruction, multiple-data (SIMD) operations that accelerate data intensive processing tasks. This processor is designed to support multiple system configurations ranging from desktop and low-end server applications, up through 4-way symmetric multiprocessor (SMP) configurations.

Each processing unit of the IBM PowerPC 970MP RISC Microprocessor consists of three main components:

- The core, which includes the VPUs
- The storage subsystem (STS), which includes the core interface logic, noncacheable unit, L2 cache and controls, and bus interface unit
- Pervasive functions

The block diagram in *Figure 1-1* on page 36 shows the major functional units comprising the core and storage subsystem. In the core, these units include instruction fetch, decode and dispatch units, plus the register files and execution units. The storage subsystem includes the second level (L2) cache and interface units.

The block diagram in *Figure 1-2* on page 37 shows how the two processing units (PU0 and PU1) are connected through the common logic to the processor interface.

Figure 1-1. 970MP Block Diagram

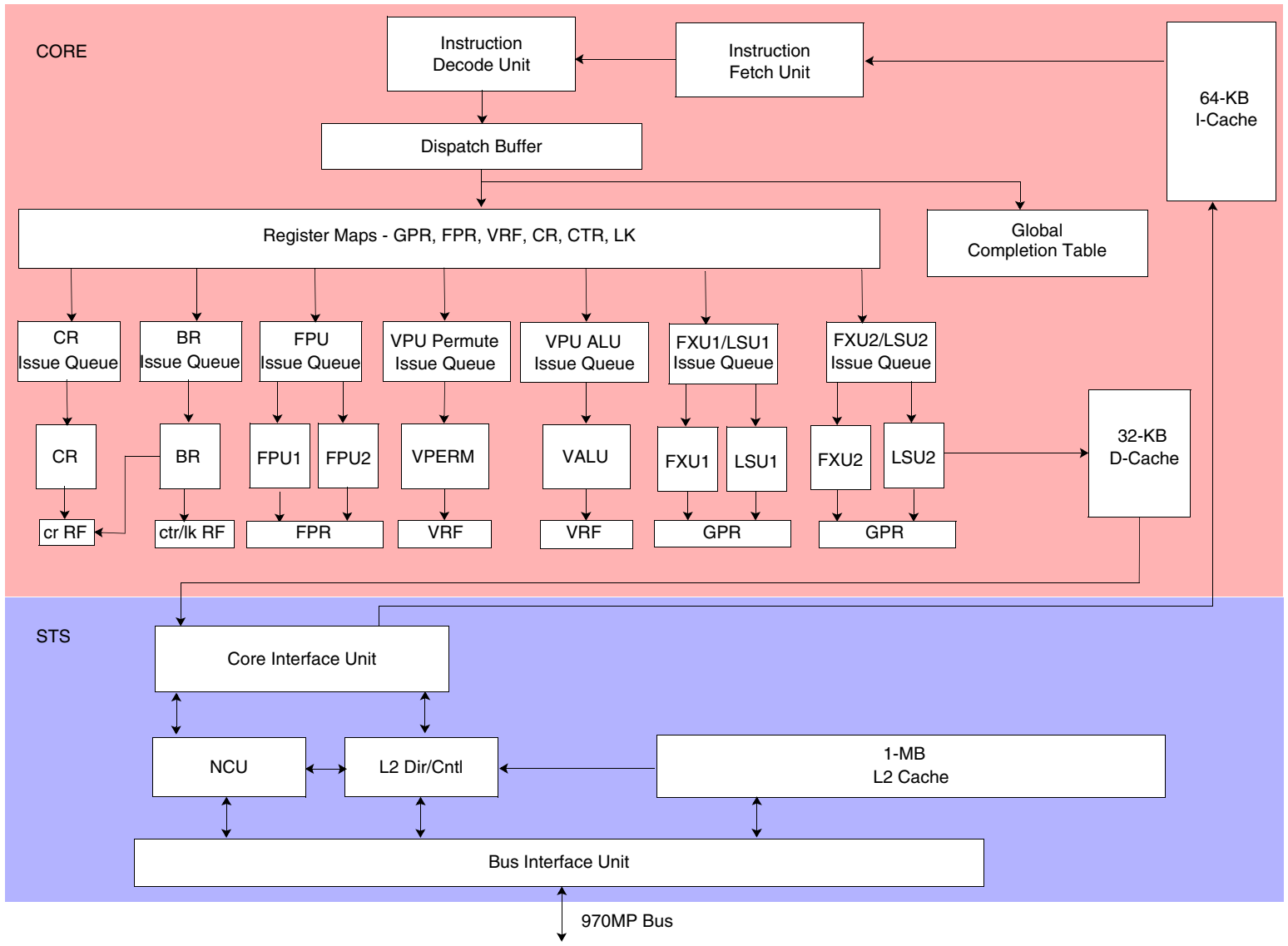
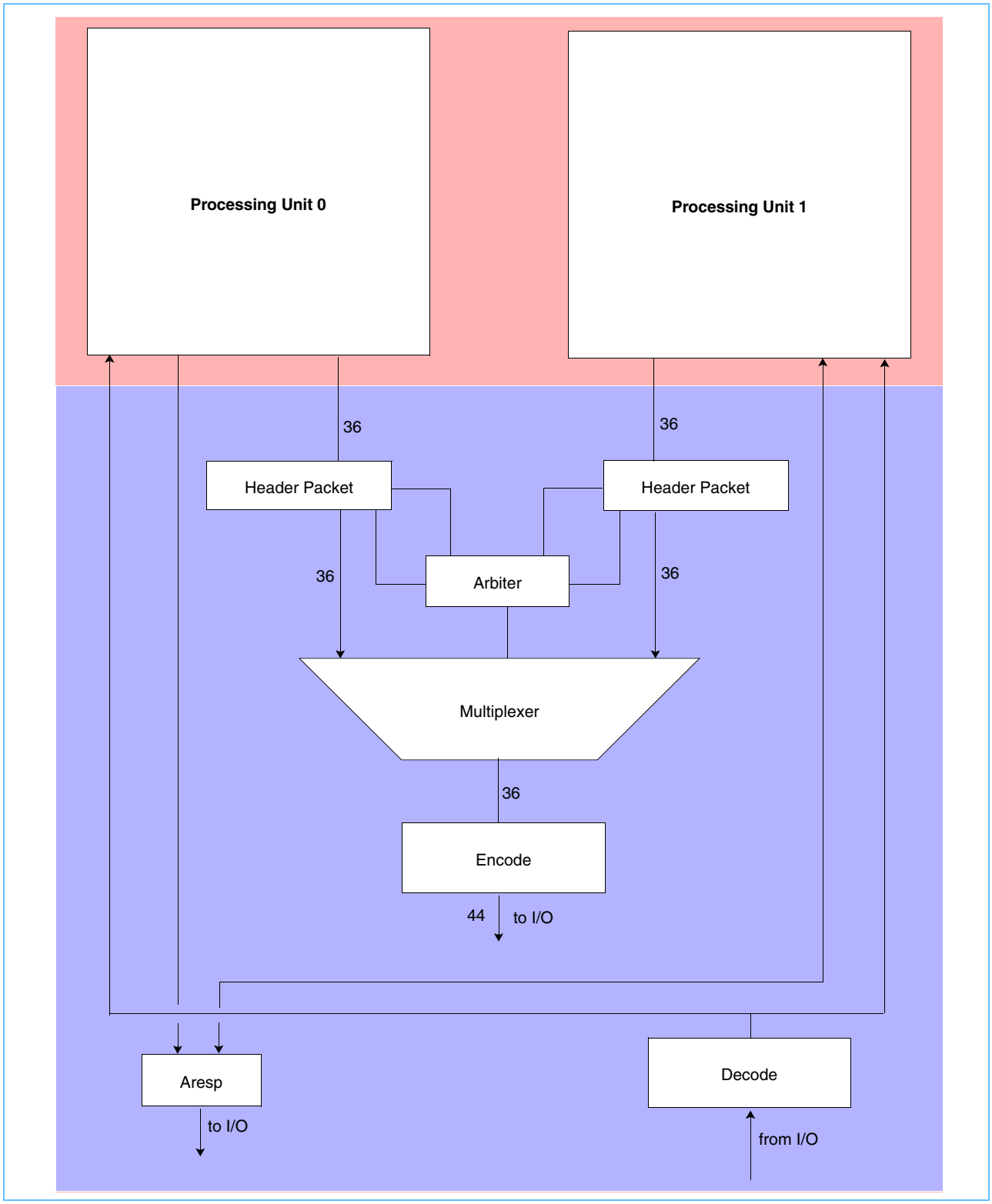


Figure 1-2. 970MP Dual Core with Common Arbitration Logic



1.2 PowerPC 970MP Functional Units

1.2.1 Introduction

This section provides an overview of the 970MP microprocessor core, VPU, storage, and bus interface units of each processing unit. It includes a summary and details of key design fundamentals.

1.2.1.1 Key Design Fundamentals of the Microprocessor Core

- 64-bit implementation of the PowerPC® Application System (AS) Architecture (version 2.01)
  - Binary compatible with all PowerPC AS application level code (user [problem] state)
  - Binary compatible with all PowerPC application level code (user [problem] state)
  - Support for the 32-bit operating system bridge facility
  - Vector/SIMD multimedia extension
- Layered implementation strategy for very high-frequency operation
  - Deeply pipelined design
    - 16 stages for most fixed-point register-to-register operations
    - 18 stages for most load-and-store operations (assuming an L1 D-cache hit)
    - 21 stages for most floating-point operations
    - 19, 22, and 25 stages for fixed-point, complex-fixed, and floating-point operations, respectively, in the vector unit arithmetic logic unit (VALU).
    - 19 stages for VPU permute operations
  - Dynamic instruction cracking[1] for some instructions allows for simpler inner core dataflow
    - Dedicated dataflow for cracking one instruction into two internal operations
    - Microcoded templates for longer emulation sequences
- Speculative superscalar inner core organization
  - Aggressive branch prediction
    - Prediction for up to two branches per cycle
    - Support for up to 16 predicted branches in flight
    - Prediction support for branch direction and branch addresses
  - In-order dispatch of up to five operations into the distributed issue queue structure
  - Out-of-order issue of up to 10 operations into 10 execution pipelines
    - Two load or store operations
    - Two fixed-point register-to-register operations
    - Two floating-point operations
    - One branch operation
    - One Condition Register operation
    - One VPU permute operation
    - One VPU arithmetic logic unit (ALU) operation
  - Register renaming on General Purpose Registers (GPRs), Floating-Point Registers (FPRs), Vector Registers (VRs), Condition Register (CR) fields, two bits of the Integer Exception Register (XER), Floating-Point Status and Control Register (FPSCR), the Vector Save/Restore Register (VRSAVE), Vector Status and Control Register (VSCR), Link Register (LR), and Count Register (CTR)

1. Process by which some complex instructions are broken into two simpler, more RISC-like instructions.

- Large number of instructions in flight (theoretical maximum of 215 instructions)
  - Up to 16 instructions in the instruction fetch unit (fetch buffer and overflow buffer)
  - Up to 32 instructions in the instruction fetch buffer in the instruction decode unit
  - Up to 35 instructions in three decode pipe stages and four dispatch buffers
  - Up to 100 instructions in the inner core (after dispatch)
  - Up to 32 stores queued in the store queue (STQ) (available for forwarding)
  - Fast, selective flush of incorrect speculative instructions and results
- Specific focus on storage latency management
  - Out-of-order and speculative issue of load operations
  - Support for up to eight outstanding L1 cache line misses
  - Hardware-initiated instruction prefetching from the L2 cache
  - Software-initiated data stream prefetching with support for up to eight active streams
  - Critical word forwarding—critical sector first
  - New branch processing—prediction hints for branch instructions
- Power management
  - Static power management
    - Software initiated Doze, Nap, and Deep Nap low-power modes
  - Dynamic power management
    - Parts of the design stop their clocks when not in use under hardware control
  - Power tuning through frequency scaling
    - Software initiated slow down of the processor; selectable to a half or quarter of the nominal operating frequency

1.2.1.2 Detailed Features of the Microprocessor Core

- Instruction fetching and branch prediction
  - 64-KB, direct-mapped instruction cache (I-cache)
    - 128-byte lines (broken into four 32-byte sectors)
    - Dedicated 32-byte read/write interface from the L2 cache with a critical-sector-first reload policy
    - Effective-address index, real address tags
    - Cache supports one read or one write per cycle
    - Five additional predecode bits per word to aid in fast decoding and group formation
    - Parity protected with a force invalidate and reload on parity error
  - 128 total entries in the effective-to-real-address translation (ERAT) cache; 2-way, set-associative
    - Organization is 64 entries by two ways
    - Each entry translates 4 KB (no large page support; large pages take multiple entries)
  - 4-entry, 128-byte, instruction prefetch queue above the I-cache; hardware-initiated prefetches
  - Fetch a 32-byte aligned block of eight instructions per cycle
  - Branch prediction
    - Scan all eight fetched instructions for branches each cycle
    - Predict up to two branches per cycle
    - 3-table prediction structure: global, local, and selector (16 K entries x 1 bit each)
    - 16-entry link stack for address prediction (with stack recovery)
    - 32-entry count cache for address prediction (indexed by the address of the Branch Conditional to Count Register [**bcctr**] instructions)

- Instruction decode and preprocessing
  - 3-cycle pipeline to decode and preprocess instructions
    - Dedicated dataflow for cracking one instruction into two internal operations
    - Microcoded templates for longer emulation sequences of internal operations
    - All internal operations expanded into 86-bit internal form to simplify subsequent processing and explicitly expose register dependencies for all register pools
    - Dispatch groups (up to five instructions) formulated along with inter-instruction dependence masks
  - Cracked and microcoded instructions have access to four renamed emulation GPRs (eGPRs), one renamed emulation FPR (eFPR), and one renamed emulation CR (eCR) field (in addition to architected facilities)
  - 8-entry (16 bytes per entry) instruction fetch buffer (up to eight instructions in and five instructions out during each cycle)
  - Microcode patch facility allows most instructions other than branches to trap to microcode, which can be programmed to either emulate the effects of the instruction or cause an interrupt.
- Instruction dispatch, sequencing, and completion control
  - Four dispatch buffers, which can hold up to four dispatch groups when the global completion table (GCT) is full
  - 20-entry global completion table
    - Group-oriented tracking associates a 5-operation dispatch group with a single GCT entry
    - Tracks internal operations from dispatch to completion for up to 100 operations
    - Capable of restoring the machine state for any of the instructions in flight
      - — Very fast restoration for instructions on group boundaries (that is, branches)
      - — Slower for instructions contained within a group
  - Supports precise exceptions (including machine check interrupt)
  - Register renaming resources
    - 80-entry GPR rename mapper (32 architected GPRs plus four eGPRs and VRSAVE)
    - 80-entry FPR rename mapper (32 architected FPRs plus one eFPR)
    - 80-entry Vector Register file (VRF) rename mapper (32 architected VRFs)
    - 24-entry XER rename mapper (the XER is broken into mappable and nonmappable fields)
      - — Two mappable fields: ov and ca
      - — Nonmappable field: string-count
    - 16-entry LR/CTR rename mapper (one architected LR and one architected CTR)
    - 32-entry CR rename mapper (eight architected CR fields plus one eCR field)
    - 20-entry FPSCR rename mapper
    - VRSAVE
    - VSCR
  - Instruction queuing resources:
    - Two 18-entry issue queues for fixed-point and load/store instructions
    - Two 10-entry issue queues for floating-point instructions
    - 12-entry issue queue for branch instructions
    - 10-entry issue queue for CR-logical instructions
    - 16-entry issue queue for vector permute instructions
    - 20-entry issue queue for vector ALU instructions and VPU stores

- Two fixed-point execution pipelines
  - Both capable of basic arithmetic, logical, and shifting operations
  - Both capable of multiplies
  - One capable of divides; the other capable of SPR operations
  - Out-of-order issue with bias towards oldest operations first
  - Symmetric forwarding between fixed-point and load/store execution pipelines
- Load/store execution pipelines
  - Two 6-stage load/store execution pipelines
  - Out-of-order issue with bias towards oldest operations first
    - Stores issue twice—an address generation operation (load/store), and a data steering operation (FXU/FPU/VPU)
  - 32-KB, 2-way, set-associative D-cache
    - Triple ported to support two reads and one write every cycle (no banking)
    - 2-cycle load-use penalty for FXU loads
    - 4-cycle load-use penalty for FPU loads
    - 3-cycle load-use penalty for loads to vector permute unit (VPERM)
    - 4-cycle load-use penalty for loads to VALU
    - Store-through policy; no allocation on store misses
    - 128-byte cache line
    - Least recently used (LRU) replacement policy
    - Dedicated 32-byte reload interface from the L2 cache
    - Effective-address index, real address tags (hardware fix up on alias cases)
    - Parity protected; precise machine check interrupt on parity error; software fix if HID5[50] equals '1'. Otherwise, recovery is done by hardware (default).
  - 128-entry (total) ERAT cache, 2-way, set-associative
    - Organization is 64 entries by two ways
    - Each entry translates 4 KB (no large page support; large pages take multiple entries)
  - 32-entry store queue logically above the D-cache (real address based; content-addressable memory [CAM] structure)
    - Store addresses and store data can be supplied on different cycles
    - Stores wait in this queue until they are completed; then they write the cache
    - Supports store forwarding to inclusive subsequent loads (even if both are speculative)
  - 32-entry load reorder queue (real address based; CAM structure)
    - Keeps track of out-of-order loads and watches for hazards
      - Previous store to the same address that gets executed after the load causes a flush
      - Previous load from the same address when a cross-invalidate has occurred causes a flush
  - 8-entry load miss queue (LMQ) (real address based)
    - Keeps track of loads that have missed in the L1 D-cache
    - Allows a second load from the same cache line to merge onto a single entry

- Branch and Condition Register execution pipelines
  - One branch execution pipeline
    - Computes actual branch address and branch direction for comparison with prediction
    - Redirects instruction fetching if either prediction was incorrect
    - Assists in training and maintaining the branch table predictors, the link stack, and the count cache
  - One Condition Register logical pipeline
    - Executes CR logical instructions and the CR movement operations
    - Executes some Move To Special Purpose Register (**mtspr**) and Move From Special Purpose Register (**mfspr**) instructions also
  - Out-of-order issue with a bias towards oldest operations first
- Floating-point execution pipelines
  - Two 9-stage floating-point execution pipelines (6-stage execution)
    - Both capable of the full set of floating-point instructions
    - All data formats supported in hardware (no floating-point assist interrupts)
  - Out-of-order issue with bias towards oldest operations first
  - Symmetric forwarding between the floating-point pipelines
  - No support for the non-IEEE mode
- VPU execution pipelines
  - Two dispatchable units:
    - VALU contains three subunits:
      - — Vector simple fixed: 1-stage execution
      - — Vector complex fixed: 4-stage execution
      - — Vector floating point: 7-stage execution
    - VPERM: 1-stage execution
  - Out-of-order issue with a bias towards oldest operations first
  - Symmetric forwarding between the permute and VALU pipelines
- Unified second-level memory management (address translation)
  - 1024-entry, 4-way, set-associative translation lookaside buffer (TLB)
    - Supports new large page architecture (16-MB large pages supported)
    - Hardware-based reload (from the L2 cache interface; no L1 D-cache impact)
    - Hardware-based update of the referenced (R) and changed (C) bits in a page table entry (PTE)
    - Parity protected; precise machine check interrupt on parity error (software fix up)
  - 64-entry fully associative segment lookaside buffer (SLB)
    - SLB miss results in an interrupt; software reload of the SLB
    - SLB can also be loaded by the 32-bit PowerPC Segment Register instructions
  - Supports a 65-bit virtual address and a 42-bit real address
- Data stream prefetch
  - Eight data prefetch streams supported in hardware. Eight hardware streams are only available if VPU prefetch instructions are disabled.
  - Four VPU prefetch streams supported using four of the eight hardware streams. The VPU prefetch mapping algorithm supports most commonly used forms of vector prefetch instructions.

1.3 970MP Dual-Core Module

The 970MP chip consists of two processing units, each containing an execution core with L1 caches, a storage subsystem including an L2 cache, and pervasive functions. In addition, a small amount of common logic that is outside either processing unit is included to connect each processing unit to the single bus interface.

Generally, the two processing units function as would two processing units on separate chips. For example, they maintain memory coherence through the North Bridge; they are able to Doze independently; they have private access to most processor resources, including their own L2 cache. Also, like processors on separate chips, they scale frequency together, using the power tuning facility.

However, sharing the same chip constrains the behavior of the two processing units in several ways. First, the two processing units have separate voltage planes for power, but the processing unit voltages will always be the same when the two processors are running. Similarly, the processing unit frequencies will always be the same. The two 970MP processing units must go into and come out of Deep Nap together.

The other difference between having dual processing units on a single chip, versus two separate chips, is that they share a single processor interface (PI) to the North Bridge. This requires that the interface between the bus interface unit (BIU) and the PI logic be enhanced with buffers and multiplexers to support the sharing of the PI between the two processing units. *Figure 1-2* on page 37 shows the relationship among the two processing units and the common logic.

For incoming PI data and commands, the output of the PI decoder is passed directly to both processors. For outgoing PI data and commands, an arbiter and multiplexer are introduced in front of the PI encoder to give one or the other processor access to the outgoing PI bus at any given time. The arbiter implements a round robin scheme, with provisions for adjusting priorities when one processing unit receives repeated serial retries. Logic in the BIU of each processing unit is modified to allow the arbiter to prevent that processing unit from sending data to the PI bus when a transaction from the other processor is in progress. The PI bus configuration parameters apply to a single bus, not to the individual processors. The arbiter enforces those parameters, such as the command pipeline delay (COMPACE) timing. To minimize dead time on the bus, header packets for each processor are queued at the arbiter. Finally, snoop responses from the two processors are combined on chip, and sent as a single response over the PI bus to the North Bridge, as indicated in the lower left corner of *Figure 1-2* on page 37.

The additional logic at the PI/BIU interface may require different values for the programmable bus delay parameters than those used for the previous design. The range of some of these parameters has therefore been increased (see *Table 11-1* on page 371).

Intercommunication between the processors on chip occurs just as if they were on separate chips, through the North Bridge. In particular, on-chip L2-to-L2 intervention is not supported.

2. Programming Model

This chapter describes the 970MP programming model, emphasizing those features specific to the 970MP microprocessor and summarizing those that are common to PowerPC processors. It consists of two major sections, which describe the following:

- Registers implemented in each 970MP processing unit
- 970MP instruction set

2.1 970MP Processor Register Set

This section describes the registers implemented in each 970MP processing unit. It includes an overview of registers defined by the PowerPC Architecture, highlighting differences in how these registers are implemented in the 970MP processing units. It also includes a detailed description of 970MP-specific registers.

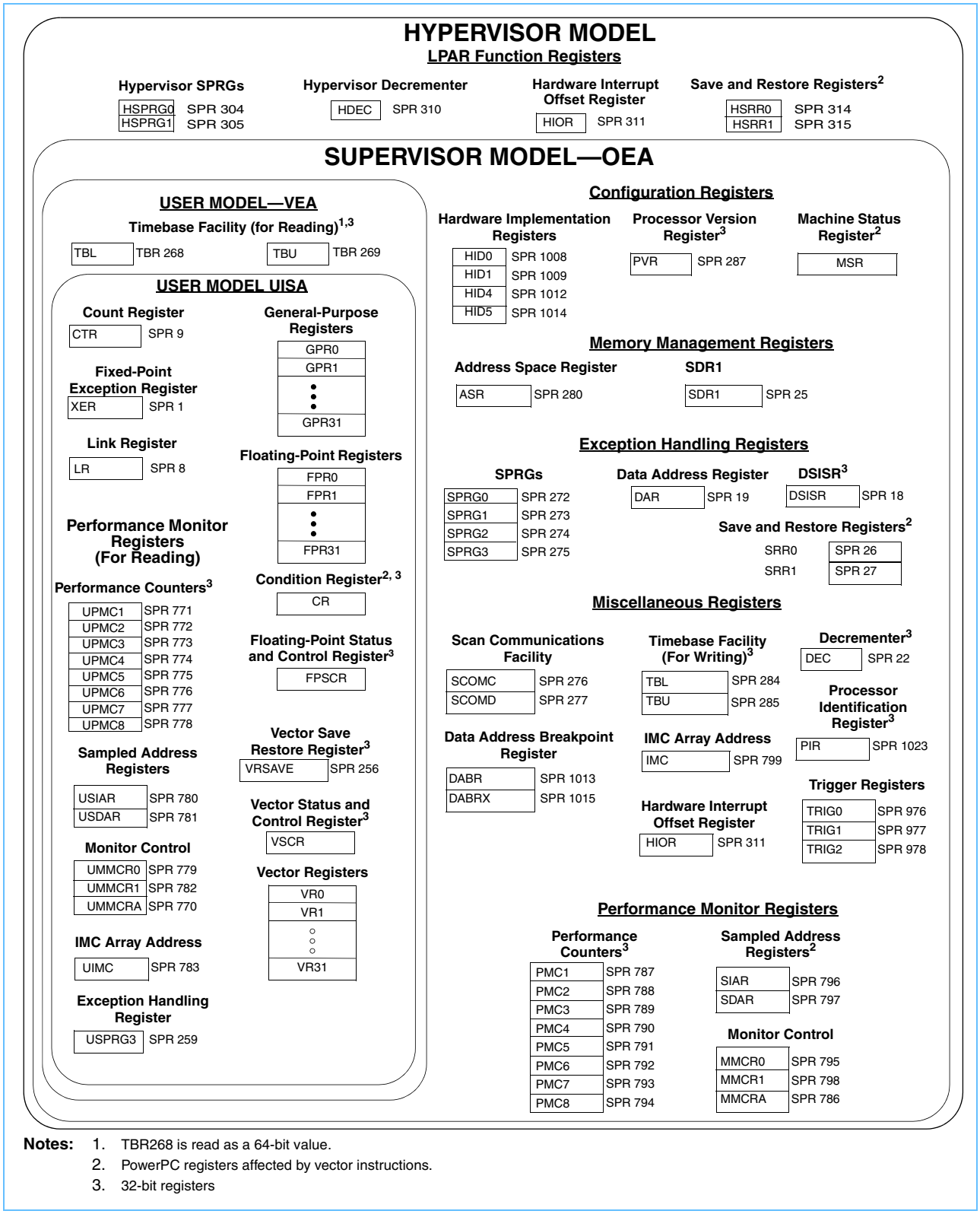
Registers are defined at all three levels of the PowerPC Architecture—user instruction set architecture (UISA), virtual environment architecture (VEA), and operating environment architecture (OEA). The PowerPC Architecture defines register-to-register operations for all computational instructions. Source data for these instructions is accessed from the on-chip registers or is provided as immediate values embedded in the opcode. The 3-register instruction format allows specification of a target register distinct from the two source registers, thus preserving the original data for use by other instructions and reducing the number of instructions required for certain operations. Data is transferred between memory and registers with explicit load-and-store instructions only.

PowerPC processors have two levels of privilege—supervisor mode of operation (typically used by the operating system) and user mode of operation (used by the application software; also called problem state). The programming models incorporate 32 General Purpose Registers (GPRs), 32 Floating-Point Registers (FPRs), 32 Vector Registers (VRs), Special-Purpose Registers (SPRs), and several miscellaneous registers. Each PowerPC microprocessor also has its own unique set of Hardware Implementation-Dependent (HID) Registers.

While running in supervisor mode, the operating system is able to execute all instructions and access all registers defined in the PowerPC Architecture. In this mode, the operating system establishes all address translations and protection mechanisms, loads all Processor State Registers, and sets up all other control mechanisms defined on the 970MP microprocessor. While running in user mode (problem state), many of these registers and facilities are not accessible. Any attempt to read or write to these registers in user mode results in a program exception.

The registers implemented on each of the 970MP processing units are shown in *Figure 2-1 970MP Programming Model—Registers* on page 46. The number to the right of the SPRs indicates the number that is used in the syntax of the instruction operands to access the register (for example, the number used to access the Integer Exception Register (XER) is SPR 1). These registers can be accessed using the Move To Special Purpose Register (**mtspr**) and Move From Special Purpose Register (**mfspr**) instructions. The inclusion of the vector processing unit (VPU) involves additional registers, and affects bit settings in some of the PowerPC registers (including the Machine State Register [MSR], Machine Status Save/Restore Register 1 [SRR1], and Condition Register [CR]) when the VPU facility is in use.

Figure 2-1. 970MP Programming Model—Registers



Notes:
1. TBR268 is read as a 64-bit value.
2. PowerPC registers affected by vector instructions.
3. 32-bit registers

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

Implementation Note: The 970MP microprocessor fully decodes the special purpose register (SPR) field of the instruction. If the SPR specified is undefined, an illegal instruction program exception occurs.

- **User-level registers** (UISA)—The user-level registers can be accessed by all software with either user or supervisor privileges. They include the following registers:
  - General-Purpose Registers (GPRs). The 32 GPRs (GPR0–GPR31) serve as data source or destination registers for fixed-point instructions and provide data for generating addresses.
  - Floating-Point Registers (FPRs). The 32 FPRs (FPR0–FPR31) serve as the data source or destination for all floating-point instructions.
  - Condition Register (CR). The 32-bit CR consists of eight 4-bit fields, CR0–CR7, that reflect results of certain arithmetic operations and provide a mechanism for testing and branching.
  - Floating-Point Status and Control Register (FPSCR). The FPSCR contains all floating-point exception signal bits, exception summary bits, exception enable bits, and rounding control bits needed for compliance with the IEEE 754 standard.
  - Vector Registers (VRs). The vector register file consists of 32 VRs (VR0-VR31). The VRs serve as vector source and vector destination registers for all vector instructions.
  - Vector Status and Control Register (VSCR). The VSCR contains the non-Java™ control bit and the saturation status bit associated with vector operations.

  The remaining user-level registers are SPRs. Note that the PowerPC Architecture provides a separate mechanism for accessing SPRs (the **mtspr** and **mfspr** instructions). These instructions are commonly used to explicitly access certain registers, while other SPRs may be more typically accessed as a side effect of executing other instructions.
  - Link Register (LR). The LR provides the branch target address for the Branch Conditional to Link Register (**bclr***x*) instruction. It can be used to hold the logical address of the instruction that follows a branch and link instruction, typically used for linking to subroutines.
  - Count Register (CTR). The CTR holds a loop count that can be decremented during execution of appropriately coded branch instructions. The CTR can also provide the branch target address for the Branch Conditional to Count Register (**bcctr***x*) instruction.
  - Vector Save/Restore Register (VRSAVE). The VRSAVE assists the application and operating system software in saving and restoring the Vector Register architectural state across context-switching events.
  - User Performance Monitor Counter Registers (UPMC1-UPMC8). UPMC1-UPMC8 provide user-level read access to the Performance Monitor Counter Registers (PMC1–PMC8).
  - User Monitor Mode Control Registers (UMMCR0, UMMCR1, UMMCRA). These registers provide user-level read access to the Monitor Mode Control Registers (MMCR0, MMCR1, MMCRA).
  - User Instruction Match Content-Addressable Memory (CAM) Register (UIMC). The UIMC provides user-level read access to the Instruction Match CAM Register (IMC).
  - User Sampled Instruction Address Register (USIAR). The USIA provides user-level read access to the Sampled Instruction Address Register (SIAR).

  - User Sampled Data Address Register (USDAR). The USDA provides user-level read access to the Sampled Data Address Register (SDAR).
  - Integer Exception Register (XER). The XER indicates overflow and carries for integer operations and the number of bytes to be transferred by the indexed instructions of the load/store string.

    Implementation Note: The architecture defines XER[44:56] as reserved.
  - Software Use Special Purpose Register 3 (SPRG3). SPRG3 can be read in problem state using SPR 259.

- **User-level registers** (VEA)—The PowerPC VEA defines the time-base facility (TB), which consists of two 32-bit registers—Time-Base Upper (TBU) and Time-Base Lower (TBL). The Time-Base Registers can be written to only by supervisor-level instructions, but can be read by both user and supervisor-level software.
- **Supervisor-level registers** (OEA)—The OEA defines the registers that an operating system uses for memory management, configuration, exception handling, and other operating system functions. The OEA defines the following supervisor-level registers:
  - Configuration registers
    - Machine State Register (MSR). The MSR defines the state of the processor. The MSR can be modified by the Move to Machine State Register (**mtmsr**), Move to Machine State Register Doubleword (**mtmsrd**), System Call (**sc**), and Return from Exception Doubleword (**rfid**) instructions. It can be read by the Move from Machine State Register (**mfmsr**) instruction. When an exception is taken, the contents of the MSR are saved to the Machine Status Save/Restore Register 1 (SRR1). See *Section 2.1.1.1 MSR Register (MSR)* on page 51 for more information.
    - Processor Version Register (PVR). This is a read-only register that identifies the version (model) and revision level of the PowerPC processor. See the *IBM PowerPC 970MP RISC Microprocessor Datasheet* for details of the PVR.
  - Memory management registers
    - Address Space Register (ASR). In the 970MP microprocessor, the Address Space Register is supported. Due to the software reload of the segment lookaside buffers (SLBs) on the 970MP microprocessor, this register does not actually participate in any other specific hardware functions on the chip. It has been included as a convenience (and performance enhancement) for the SLB reload software.
    - Storage Description Register (SDR1). SDR1 specifies the page-table base address used in virtual-to-physical address translation.
  - Exception-handling registers
    - Data Address Register (DAR). After a data storage interrupt (DSI) exception or an alignment exception, the DAR is set to the effective address (EA) generated by the faulting instruction.
    - Software Use Special Purpose Registers 0 - 3 (SPRG0–SPRG3). SPRG0–SPRG3 are provided for operating system use.
    - Data Storage Interrupt Status Register (DSISR). DSISR defines the cause of DSI and alignment exceptions.
    - Machine Status Save and Restore Register 0 (SRR0). SRR0 is used to save the address of the instruction at which execution continues when **rfid** executes at the end of an exception handler routine. See *Section 2.1.1.2 Machine Status Save/Restore Register (SRR1)* on page 51 for more information.

- Machine Status Save and Restore Register 1 (SRR1). SRR1 is a 64-bit register used to save machine status on exceptions and restore the Machine Status Register when an **rfid** instruction is executed. See *Section 2.1.1.2 Machine Status Save/Restore Register (SRR1)* on page 51 for more information.

  Note: For information about how specific exceptions affect SRR1, see the individual exception in *Section 4.5 Exception Definitions* on page 138.

– Miscellaneous registers

- Time Base (TB). This register is a 64-bit structure provided for maintaining the time of day and operating interval timers. The TB consists of two 32-bit registers—Time-Base Upper (TBU) and Time-Base Lower (TBL). The Time-Base Registers can be written to only by supervisor-level software, but can be read by both user- and supervisor-level software. See *Section 2.1.1.3 Time Base and Decrementer (TB, DEC)* on page 52 for more information.

  Implementation Note: In the 970MP microprocessor, the Time-Base Register is incremented once every sixteen full frequency processor clocks. Alternatively, when HID0[19] is set to '1', the Time-Base Register is incremented at the input frequency of the timebase_enable input pin (TBEN).

- Decrementer Register (DEC). This register is a 32-bit decrementing counter that provides a mechanism for causing a decrementer exception after a programmable delay. See *Section 2.1.1.3 Time Base and Decrementer (TB, DEC)* on page 52 for more information.

  Implementation Note: In the 970MP microprocessor, the Decrementer Register is decremented once every 16 full frequency processor clocks. Alternatively, when HID0[19] is set to '1', the Decrementer Register is decremented at the TBEN input frequency.

- Processor ID Register (PIR). The PIR Register is used to differentiate between individual processors in a multiprocessor environment. See *Section 2.1.1.4 Processor ID Register (PIR)* on page 52 for more information.

– Performance Monitor Registers. The following registers are used to define and count events for use by the performance monitor:

- The Performance Monitor Counter Registers (PMC1–PMC8) are used to record the number of times a certain event has occurred. See *Section 2.1.2.6 Performance Monitor Registers (MMCR0, MMCR1, MMCRA, PMC1-8)* on page 65 for more information.

- The Monitor Mode Control Registers (MMCR0, MMCR1, MMCRA) are used to identify what events will be monitored and to enable various performance monitor interrupt functions. See *Section 2.1.2.6 Performance Monitor Registers (MMCR0, MMCR1, MMCRA, PMC1-8)* on page 65 for more information.

- The Sampled Instruction Address Register (SIAR) contains the effective address of an instruction executing at or around the time that the processor signals the performance-monitor interrupt condition. See *Section 2.1.2.7 Sampled Instruction Address and Sampled Data Address Registers (SIAR, SDAR)* on page 66 for more information.

- The Sampled Data Address Register (SDAR) contains the effective address of the storage access instruction executing at or around the time that the processor signals the performance monitor interrupt condition. See *Section 2.1.2.7 Sampled Instruction Address and Sampled Data Address Registers (SIAR, SDAR)* on page 66 for more information.

- **970MP-specific registers**—The PowerPC Architecture allows implementation- specific SPRs. The following registers are incorporated in each 970MP processing unit:

  Note: In the 970MP microprocessor, these registers are all supervisor-level registers.

  – Hardware Implementation-Dependent Register 0 (HID0). This register controls various functions, such as enabling checkstop[1] conditions, locking, enabling, invalidating the instruction and data caches, and power modes. See *Section 2.1.2.2 HID Registers (HID0, HID1, HID4, and HID5)* on page 56 for more information.

  – Hardware Implementation-Dependent Register 1 (HID1). HID1 contains additional mode bits that are related to the instruction fetch and instruction decode functions in the 970MP microprocessor. See *Section 2.1.2.2 HID Registers (HID0, HID1, HID4, and HID5)* on page 56 for more information.

  – Hardware Implementation-Dependent Register 4 (HID4). HID4 contains bits related to the load/store function in the 970MP microprocessor. See *Section 2.1.2.2 HID Registers (HID0, HID1, HID4, and HID5)* on page 56 for more information.

  – Hardware Implementation-Dependent Register 5 (HID5). HID5 contains bits related to the load/store function in the 970MP microprocessor. See *Section 2.1.2.2 HID Registers (HID0, HID1, HID4, and HID5)* on page 56 for more information.

  – Data Address Breakpoint Register (DABR) and Data Address Breakpoint Register Extension (DABRX). The DABR controls the data address breakpoint mechanism, which provides a means of detecting load-and-store accesses to a designated double word. See *Section 2.1.2.3 Data Address Breakpoint Register (DABR)* on page 63 for more information.

  – Scan Communications Register (SCOMC). SCOMC is a control register that includes a command field, a destination field, and a set of status bits. See *Section 2.1.2.8 Scan Communication Registers (SCOMC and SCOMD)* on page 66 for more information.

  – Scan Communications Register (SCOMD). SCOMD is an associated data register that acts as either a source of data or as a destination for data depending on the command placed into the SCOMC. See *Section 2.1.2.8 Scan Communication Registers (SCOMC and SCOMD)* on page 66 for more information.

  – Instruction Match CAM Registers (IMCs). The IMC SPRs are used to access the IMC array, which contains the mask values used for instruction matching. The **mtimc** and **mfimc** instructions can be executed only in supervisor mode. See *Section 2.1.2.5 Instruction Match CAM Array Access Register (IMC)* on page 64 for more information.

  – Trigger Registers (TRIG0-TRIG2). Writes to the Trigger Registers, named TRIG0, TRIG1, and TRIG2, can be inserted in the instruction stream to cause triggers to the on-chip trace array debug logic. These are intended to be used for debug and bring-up only and architecturally behave as no-ops. See *Section 2.1.2.12 Trigger Registers (TRIG0, TRIG1, TRIG2)* on page 67 for more information.

  – Hardware Interrupt Offset Register (HIOR). The HIOR is used for interrupt vector relocation. See *Section 2.1.2.13 Hardware Interrupt Offset Register (HIOR)* on page 68 for more information.

Note: While it is not guaranteed that the implementation of 970MP-specific registers is consistent among PowerPC processors, other processors may implement similar or identical registers.

1. Hardware has detected a condition that it cannot resolve, and which prevents normal operation. It stops executing instructions, responding to interrupts, and so on.

2.1.1 Architected Registers in the 970MP Implementation

Several architected registers are implemented in each 970MP processing unit in a way that varies from, or extends, the definition in the PowerPC Application System (AS) architecture.

2.1.1.1 MSR Register (MSR)

The PowerPC Architecture describes the MSR bits 2, 4:47, 57, 60, and 63 as either optional or reserved. In the 970MP microprocessor, bit 38 is used as the vector processor available (VP) enable and bit 45 is used as the power management (POW) enable. The other bits are not implemented and will return the value '0' when read.

Note: Little-endian mode is not supported (that is, MSR[LE] and MSR[ILE] are treated as reserved).

Implementation Note: *Table 2-1* describes MSR bits that the 970MP microprocessor implements that deviate from the PowerPC Architecture.

Table 2-1. MSR Bits

Bit	Name	Description
3	—	Reserved; returns a value of '1' when read.
38	VP	VP available. 0 The processor prevents execution of all vector instructions including loads, stores, and moves. If such execution is attempted, a VP unavailable exception is raised. 1 The processor can execute all vector instructions. The VRSAVE Register is not protected by MSR [VP]. The data streaming family of instructions (**dst**, **dstt**, **dstst**, **dststt**, **dss**, and **dssall**) are not affected by the MSR[VP].
45	POW	Activates power management. The 970MP microprocessor will clear the POW bit when it leaves a power saving mode. See *Chapter 9 Power and Thermal Management* for more information.
47	—	Reserved. The ILE bit is not implemented in the 970MP microprocessor.
48	EE	External interrupt enable 0 The processor delays recognition of external interrupts and decrementer exception conditions. 1 The processor is enabled to take an external interrupt or the decrementer exception. **Note:** Resetting MSR[EE] masks not only the architecture-defined external interrupt and decrementer exceptions, but also the 970MP-specific instrumentation, debug, and performance monitor exceptions.
63	—	Reserved. The LE bit is not implemented in the 970MP microprocessor.

2.1.1.2 Machine Status Save/Restore Register (SRR1)

This register is used to save machine status during interrupts. In the 970MP microprocessor, SRR1 bits 1:2, 4:32, 37, 39:41, 47, 56:57, 60, and 63 are treated as reserved. These bits are not implemented and will return the value '0' when read. See *Section 4.3.2 Machine Status Save/Restore Register 1 (SRR1)* on page 133 for additional information.

Table 2-2. Additional SRR1 Bit

Bit	Function
33	SIAR and SDAR contents synchronized.

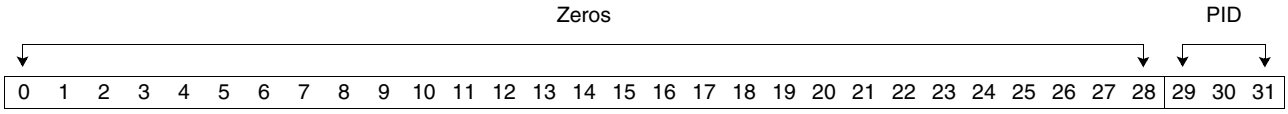
2.1.1.3 Time Base and Decrementer (TB, DEC)

The time-base counter and the decrementer are clocked at 1/16 (one-sixteenth) of the full frequency processor. The 970MP microprocessor supports two modes of operation (controlled by HID0[19] and the time-base enable input pin) for updating the time base and decrementer. When HID0[19] is zero, then the counters constantly update as long as the TBEN is high (traditional mode of operation). When HID0[19] is one, the counters update only on the rising edge of the TBEN input pin.

When the processor is stopped due to various breakpoint, debug and support processor functions, an additional mode bit, HID0[18], determines whether the time base and the decrementer continue counting. Note that some support processor operations require the use of an alternate clocking mode for scan, and in these cases, the time base and the decrementer will not continue counting.

2.1.1.4 Processor ID Register (PIR)

The Processor Identification Register (PIR) is a 32-bit register that holds a processor identification tag. In the 970MP processing unit, this tag is in the three least-significant bits (29:31). The least-significant bit of the processor identification tag (PID) is hardwired to '0' for PU0 and to '1' for PU1. This tag is used to tag bus transactions and to differentiate processors in multiprocessor systems. The PIR is a read-only register. The format of the register is as follows:



Bits	Field Name	Description
0:28	—	Reserved (read as zeros)
29:31	PID	3-bit processor ID value (least-significant bit hardwired to differentiate PU0 and PU1)

During power-on reset, PID is set to a unique value for each processor in a multi processor system. For more information about the power-on reset configuration process, see *Section 11.3.1 Initialization at Power-On Reset* on page 379.

2.1.2 PowerPC 970MP-Specific Registers

This section describes registers that are defined for the 970MP microprocessor, but are not included in the PowerPC Architecture.

2.1.2.1 Move To and Move From System Register Instructions

The 970MP architecture defines several new implementation-specific system registers. Note that some of these registers are also readable in user mode through a second set of SPR encodings, and that some of these registers have special software synchronization requirements.

The encoded SPR values for these implementation-specific registers are shown in *Table 2-3*. Note that the SPR is encoded in the **mfspr** and **mtspr** instructions. Bits 5:9 of the SPR field represent the 5 high-order bits of the SPR number, and bits 0:4 of the SPR field represent the 5 low-order bits of the SPR number.

Table 2-3. Implementation-Specific SPRs

SPR			Register Name	R/W	Synchronization Requirements		
Decimal (privileged)	Decimal (user)	SPR(5:9) SPR(0:4)			Before Reads	After Writes	Before Writes
1023		11111 11111	PIR	R	none	N/A	N/A
1013		11111 10101	DABR	R/W	none	context synchronizing instruction (CSI)	sync
1015		11111 10111	DABRX	R/W			
1008		11111 10000	HID0	R/W	none	Note 1	Note 1
1009		11111 10001	HID1	R/W	none	Note 2	Note 2
1012		11111 10100	HID4	R/W	none	Note 3	Note 3
1014		11111 10110	HID5	R/W	none	Note 4	Note 4
795	779	11000 n1011	MMCR0	R/W	none	Note 5	Note 5
798	782	11000 n1110	MMCR1	R/W	none	Note 5	Note 5
786	770	11000 n0010	MMCRA	R/W	none	Note 5	Note 5
787	771	11000 n0011	PMC1	R/W	sync	none	none
788	772	11000 n0100	PMC2	R/W	sync	none	none
789	773	11000 n0101	PMC3	R/W	sync	none	none
790	774	11000 n0110	PMC4	R/W	sync	none	none
791	775	11000 n0111	PMC5	R/W	sync	none	none
792	776	11000 n1000	PMC6	R/W	sync	none	none
793	777	11000 n1001	PMC7	R/W	sync	none	none
794	778	11000 n1010	PMC8	R/W	sync	none	none
276		01000 10100	SCOMC	R/W	none	CSI	none
277		01000 10101	SCOMD	R/W	none	CSI	none
796	780	11000 n1100	SIAR	R/W	sync	none	none
797	781	11000 n1101	SDAR	R/W	sync	none	none
799	783	11000 n1111	IMC	R/W	none	CSI	none
976		11110 10000	TRIG0	W	N/A	none	none
977		11110 10001	TRIG1	W	N/A	none	none
978		11110 10010	TRIG2	W	N/A	none	none
256		01000 00000	VRSAVE	R/W	N/A	none	none
311		01001 10111	HIOR	R/W			

Table 2-3. Implementation-Specific SPRs (Continued)

For **mtspr**, n must be '1'. For **mfspr**, reading the SPR is privileged if and only if n equals '1'.

Notes:

1. The following sequence must be used when modifying HID0:
   sync
   mtspr HID0,Rx
   mfspr Rx,HID0
   mfspr Rx,HID0
   mfspr Rx,HID0
   mfspr Rx,HID0
   mfspr Rx,HID0
   mfspr Rx,HID0
   After modifying HID0, executing six **mfspr** instructions specifying HID0 as the source and specifying the same target General Purpose Register (GPR) (Rx) in all six instructions is necessary to ensure that the modification is effective and that the processor is in a valid state to continue executing subsequent instructions.
2. The following sequence must be used when modifying HID1:
   mtspr HID1,Rx
   mtspr HID1,Rx
   isync
   Executing two **mtspr** instructions is necessary to ensure that updates to all portions of HID1 will be complete before the Instruction Cache Synchronize (**isync**) instruction completes.
3. The following sequence must be used when modifying HID4:
   sync
   mtspr HID4,Rx
   isync
   When HID4[23] is changed, the above sequence should be preceded by a Move to Segment Register (**mtsr**) and Synchronize (**sync**) instruction**,** which will cause the effective-to-real-address translations (ERATs) to be flushed.
4. The following sequence must be used when modifying HID5:
   sync
   mtspr HID5,Rx
   isync
   Whenever HID5[56] or HID5[57] is changed, the entire instruction cache must be flushed to ensure that any succeeding Data Cache Block Set to Zero (**dcbz**) instruction is executed in the context of the new HID5 bit settings.
5. Although it is not necessary to use synchronizing instructions when modifying the MMCR(0,1,A) registers, it is recommended that the following sequence be used:
   sync
   mtspr MMCRz,Rx
   isync

Table 2-4 describes the behavior of the 970MP microprocessor for the **mtspr** and **mfspr** instructions.

Table 2-4. Move To/Move From SPR Behavior

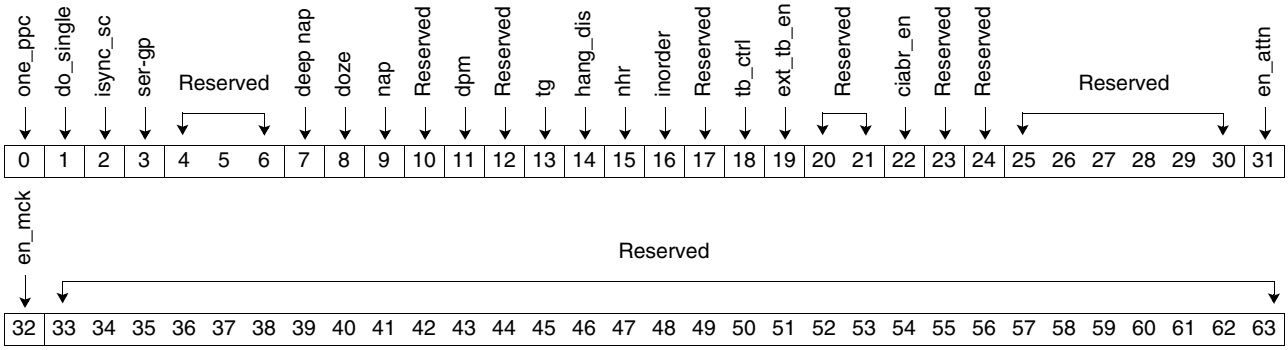
Condition				Resulting Action
SPR		MSR[PR]	R/W	
SPR(0)	Register			
1	Any invalid SPR encoding	0	**mfspr**	No-op (target register is unchanged)
1	Any invalid SPR encoding	0	**mtspr**	No action (write is inhibited)
1	Address Compare Control Register (ACCR), ASR, Control Register (CTRL), DABR, DAR, DEC, DSISR, HID0, HID1, HID4, HID5, IMC, SCOMC, SCOMD, SDR1, SDAR, SIAR, SRR0, SRR1, SPRG0, SPRG1, SPRG2, SPRG3, TBL, TBU, Performance Monitor Registers	0	**mfspr**	Returns a value to a GPR.
		0	**mtspr**	Target SPR is updated.
1	TRIG0, TRIG1, TRIG2	0	**mfspr**	Causes an illegal instruction type of program interrupt.
		0	**mtspr**	Causes a trigger to the trace array debug logic.
1	PIR	0	**mfspr**	Returns a value to a GPR.
		0	**mtspr**	Causes an illegal instruction type of program interrupt.
1	Any SPR encoding (with SPR(0) equal to '1')	1	**mtspr** **mfspr**	Causes a privileged instruction type of program interrupt.
0	Any invalid SPR encoding except: spr(0:9) = '00000 00000' spr(0:9) = '00100 00000' spr(0:9) = '00101 00000' spr(0:9) = '00110 00000'	X	**mfspr**	No-op (target register is unchanged)
		X	**mtspr**	No action (write is inhibited)
0	spr(0:9) = '00000 00000' spr(0:9) = '00100 00000' spr(0:9) = '00101 00000' spr(0:9) = '00110 00000'	X	**mtspr** **mfspr**	Causes an illegal instruction type of program interrupt.

2.1.2.2 HID Registers (HID0, HID1, HID4, and HID5)

The 970MP microprocessor includes many implementation-dependent mode bits that allow various features of the chip to be enabled and disabled. These bits are included in the Hardware Implementation-Dependent Registers (HID0, HID1, HID4, and HID5). In general, HID0 attempts to line up the 970MP microprocessor modes with the relevant ones from earlier PowerPC implementations and then adds a few new ones. HID1 contains additional mode bits that are related to the instruction fetch and instruction decode functions in the 970MP microprocessor. HID4 and HID5 contain bits related to the load/store function in the 970MP microprocessor. All of these registers are supervisor resources.

The state of each of the HID Registers after a normal scan-based POR is all zeros. The preferred state of these registers for optimal performance and function is also all zeros, except where indicated.

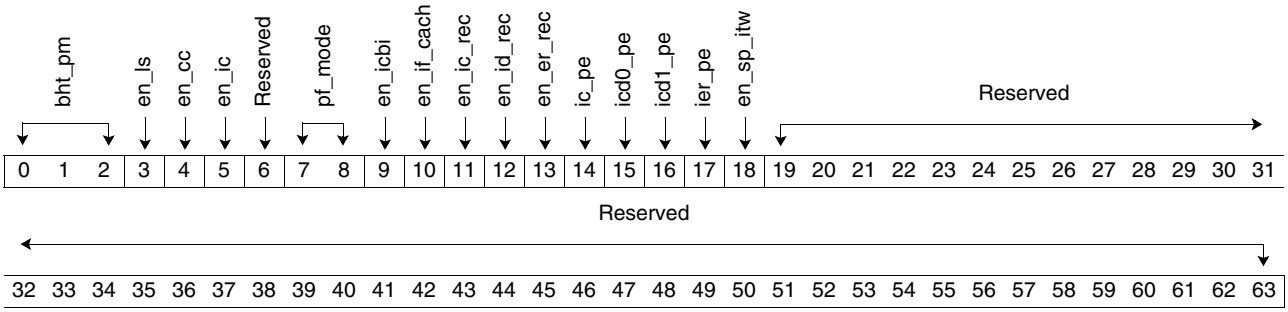
HID0 Bit Functions



Bits	Field Name	Description
0	one_ppc	One PowerPC AS instruction per dispatch group mode. An instruction may span more than one group.
1	do_single	Single group completion mode. Flush and refetch after the completion of each group or the completion of each microcoded instruction, if the instruction spans multiple groups.
2	isync_sc	Disable isync scoreboard optimization.
3	ser-gp	Serialize group dispatch. The next group is not dispatched until the previous group completes.
4:6	—	Reserved
7	deep nap	Deep nap
8	doze	Doze
9	nap	Nap
10	—	Reserved
11	dpm	Enable dynamic power management.
12	—	Reserved
13	tg	Performance monitor threshold granularity control.
14	hang_dis	Disable processor hang-detection mechanism.
15	nhr	Not hard reset. Check after snoop response in (SRI) to see if hard or soft.
16	inorder	Serialized group issue mode. The next group is not issued until the previous group completes. Does not include branch or CR-logical instructions.
17	—	Reserved

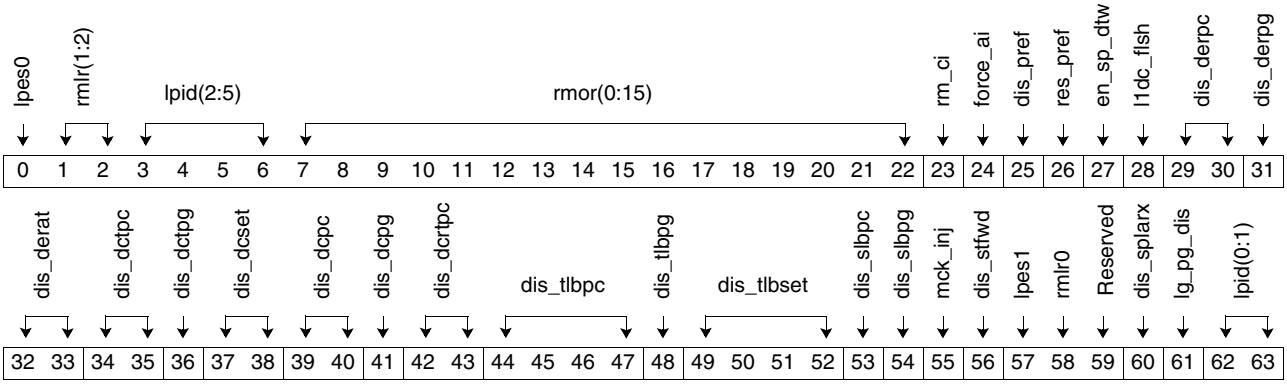
Bits	Field Name	Description
18	tb_ctrl	Enable time-base counting when the processor is stopped.
19	ext_tb_en	External time-base enable. 0 Use TBEN input as enable. TB is clocked at 1/8 of the full processor frequency. 1 Use TBEN input to clock time base (external clock).
20:21	—	Reserved
22	ciabr_en	Enable Completion Instruction Address Breakpoint Register (CIABR).
23	hdice_en	Enable hypervisor decrementer interrupt conditionally (HDICE). The initial reset value must be '0' and disables hypervisor interrupts.
24:30	—	Reserved
31	en_attn	Enable support processor attention instruction.
32	en_mck	Enable external machine check interrupts (preferred state equals '1').
33:63	—	Reserved

HID1 Bit Functions



Bits	Field Name	Description
0:2	bht_pm	Branch history table (BHT) prediction mode. 000 Static prediction 001 Unused (same as 000) 010 Global BHT prediction only 011 Global prediction with history compression 100 Local BHT prediction only 101 Unused (same as 100) 110 Full global/local prediction with global selection (gsel) 111 Full global/local prediction with gsel and history compression (preferred state)
3	en_ls	Enable link stack (preferred state equals '1').
4	en_cc	Enable count cache (preferred state equals '1').
5	en_ic	Enable instruction cache (must be '1' for proper functioning).
6	—	Reserved
7:8	pf_mode	Prefetch mode. 00 No instruction prefetch. 01 Select next sequential address (NSA) instruction prefetch. 10 Select NSA and NSA + 1 instruction prefetch (preferred state). 11 Disable prefetch buffer.
9	en_icbi	Enable forced Instruction Cache Block Invalidate (**icbi**) match mode.
10	en_if_cach	Enable instruction fetch cacheability control. 0 All instruction fetch accesses are treated as cache inhibited regardless of the state of the I bit in the page table. 1 Instruction fetch cacheability is controlled by the state of the I bit in the page table (preferred state).
11	en_ic_rec	Enable I-cache parity error recovery (preferred state equals '1').
12	en_id_rec	Enable I-directory parity error recovery (preferred state equals '1').
13	en_er_rec	Enable instruction ERAT (I-ERAT) parity error recovery (preferred state equals '1').
14	ic_pe	Force instruction cache parity error (error inject).
15	icd0_pe	Force instruction cache directory 0 parity error (error inject).
16	icd1_pe	Force instruction cache directory 1 parity error (error inject).
17	ier_pe	Force I-ERAT parity error (error inject).
18	en_sp_itw	Enable speculative tablewalks. The ERAT is never loaded using a page table entry (PTE) if PTE[G] is set to '1' (preferred state equals '1').
19:63	—	Reserved

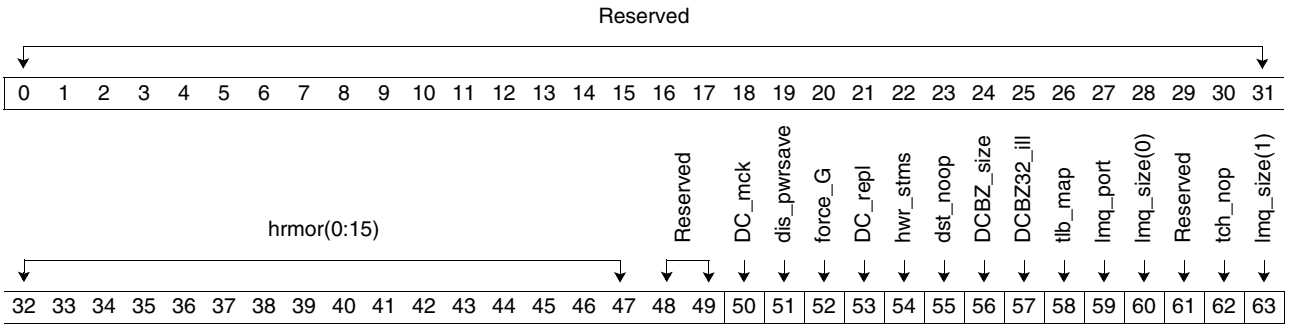
HID4 Bit Functions



Bits	Field Name	Description
0	lpes0	LPAR environment selector bit [0]. LPES[0:1] are located in HID4[0, 57]. LPES[0:1] determine how MSR[HV] is set using interrupts and how memory access is performed when not in hypervisor mode. Thi s is described in the PowerPC Architecture version 2.01.
1:2	rmlr(1:2)	LPAR real mode limit register (see HID4[58] for bit [0]).
3:6	lpid(2:5)	LPAR partition identity bits [2:5] (see also bits [62:63] for lpid(0:1).
7:22	rmor(0:15)	LPAR real mode offset register [0:15].
23	rm_ci	Data accesses in real mode are treated as cache-inhibited.
24	force_ai	Force alignment interrupt instead of microcoding unaligned operations (that is instead of breaking unaligned operations into multiple smaller operations).
25	dis_pref	Disables data prefetching.
26	res_pref	Setting HID4[26] to '1' resets the data prefetch mechanism, suppressing subsequent prefetch requests and clearing the stream detection logic, therefore stream detection is not affected by accesses performed before setting the bit back to '0'.
27	en_sp_dtw	Enable speculative load tablewalk.
28	l1dc_flsh	L1 data cache flash invalidate. 0 Normal operation 1 All sectors set to invalid and held invalid
29:30	dis_derpc	Disable data ERAT (D-ERAT) parity checking (one bit for each set).
31	dis_derpg	Disable D-ERAT parity generation (force parity to '0' on EA[0:45] only).
32:33	dis_derat	Disable one or more ways of the 4-way set associative D-ERAT (one bit per set); valid states are 00, 01, and 10.
34:35	dis_dctpc	Disable data cache tag parity checking (one bit for each set).
36	dis_dctpg	Disable data cache tag parity generation.
37:38	dis_dcset	Disable data cache set (one bit for each set).
39:40	dis_dcpc	Disable parity checking in one or more ways of the 4-way set-associative data cache (one bit per set).
41	dis_dcpg	Disable data cache parity generation.
42:43	dis_dcrtpc	Disable parity checking on the physical address tag of the data cache (one bit per set).
44:47	dis_tlbpc	Disable parity checking in one or more ways of the 4-way set-associative translation lookaside buffer (TLB) (one bit per set).
48	dis_tlbpg	Disable TLB parity generation.

Bits	Field Name	Description
49:52	dis_tlbset	Disable set in one or more ways of the 4-way set associative TLB (one bit per set); valid states are x'0', x'7', x'B', x'D', and x'E'.
53	dis_slbpc	Disable SLB parity checking.
54	dis_slbpg	Disable SLB parity generation.
55	mck_inj	Enable machine-check error injection.
56	dis_stfwd	Disable store forwarding (cause reject).
57	lpes1	LPAR environment selector bit [1]. LPES[0:1] are located in HID4[0, 57]. LPES[0:1] determine how MSR[HV] is set using interrupts and how memory access is performed when not in hypervisor mode. Thi s is described in the PowerPC Architecture version 2.01.
58	rmlr0	HID4 bits [58, 1:2] are real mode limit register bits [0:2]. 011 64 MB 111 128 MB 100 256 MB x10 1 GB x01 16 GB 000 256 GB
59	—	Reserved
60	dis_splarx	Disable speculative Load Word and Reserve Indexed (**lwarx**) and Load Double Word and Reserve Indexed (**ldarx**) instructions.
61	lg_pg_dis	Disable large page support. The large page (L) bit input to SLB will be forced to zero (software will read a zero L bit).
62:63	lpid(0:1)	LPAR partition identity bits [0:1]. HID4[62:63, 3:6] are LPID[0:5] respectively.

HID5 Bit Functions



Bits	Field Name	Description
0:31	—	Reserved
32:47	hrmor(0:15)	LPAR hypervisor real mode offset register.
48:49	—	Reserved
50	DC_mck	Machine check enabled for data cache and data cache tag parity errors (software recovery enabled).
51	dis_pwrsave	L1 data cache (D-cache), L1 D-cache tag, D-ERAT power savings disable.
52	force_G	Force guarded (G equals '1') load.
53	DC_repl	Data cache replacement algorithm. 0 Least recently used (LRU) (default) 1 First-in-first-out (FIFO)
54	hwr_stms	Number of available hardware prefetch streams. 0 Four hardware streams and four VPU streams 1 Eight hardware streams (HID5[55] must also be '1')
55	dst_noop	Data Stream Touch (DST) instructions no-op. 0 DSTs are enabled. 1 DSTs are a no-ops and discarded in the load/store unit (LSU).
56	DCBZ_size	Makes **dcbz** a 32-byte store when bit 10 of the **dcbz** instruction is set to '0'.
57	DCBZ32_ill	Makes a **dcbz** instruction with bit 10 equal to '0' an illegal instruction.
58	tlb_map	TLB mapping. 0 4-way set associative 1 Direct mapped **Note:** When setting HID5[58] to make the TLB direct mapped, the TLB set disable bits, HID4[49:52], must be cleared; otherwise, translation will not work.
59	lmq_port	Demand miss (load miss queue [LMQ] to 970MP storage subsystem [STS]). 0 Permit two per cycle. 1 Permit only one per cycle (this setting is not currently supported).
60	lmq_size(0)	Number of outstanding requests to STS. HID5 [60, 63] — Maximum outstanding requests 00 — 8 01 — 1 (this setting is not currently supported) 10 — 2 11 — 4
61	—	Reserved

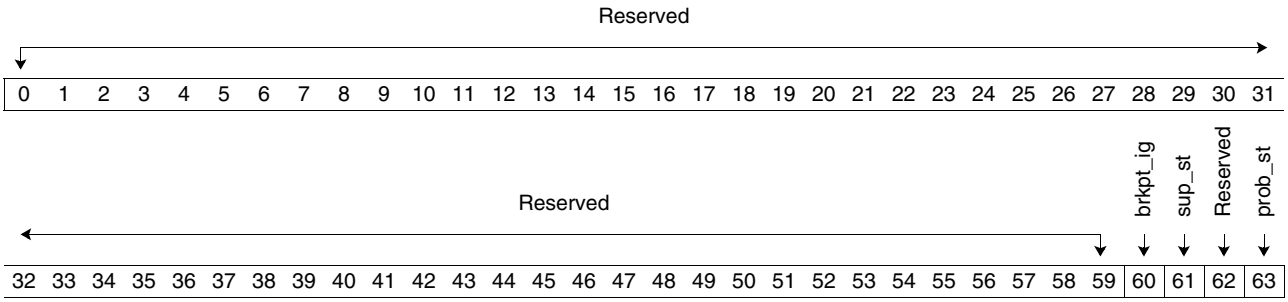
Bits	Field Name	Description
62	tch_nop	Make the Data Cache Block Touch (**dcbt)** and Data Cache Block Touch for Store (**dcbtst**) instructions act like no-ops.
63	lmq_size(1)	See description of HID5[60].

2.1.2.3 Data Address Breakpoint Register (DABR)

The data address breakpoint facility provides a means of detecting load-and-store accesses to a designated double word. The address comparison is done on an effective address. The data address breakpoint facility is controlled by the architected Data Address Breakpoint Register (DABR) and the 970MP microprocessor-specific Data Address Breakpoint Register Extension (DABRX).

Data Address Breakpoint Register Extension (DABRX)

The DABRX register is only active in hypervisor mode.



Data Address Compare

The 970MP microprocessor supports the address compare control facility and the Address Compare Control Register (ACCR) as defined in the architecture. In addition, the 970MP microprocessor supports the optional data address breakpoint facility and associated Data Address Breakpoint Register (DABR) described in the architecture. In either case, upon taking a data storage interrupt, the 970MP processing unit sets the DAR correctly.

The architecture allows some flexibility on whether an ACCR match, a DABR match, or both actually occurs for certain conditions. More specifically, in the 970MP processing unit, store conditional instructions that are executed but not successful (that is, the store does not actually occur) will cause either an ACCR match or a DABR match if the appropriate match conditions are met. String instructions with zero length will not cause ACCR or DABR matches. The **dcbz** instruction will cause a DABR match if the appropriate match conditions are met.

As an alternative to causing an interrupt, a DABR match can be made to cause various forms of hard stops or soft stops for use as a debug aid (these controls are available through special SCOM commands). In general, this capability is not recommended for use in normal system operation because it may require the presence of an engineering support processor to restart the processing unit.

2.1.2.4 Instruction Address Breakpoint Register (IABR)

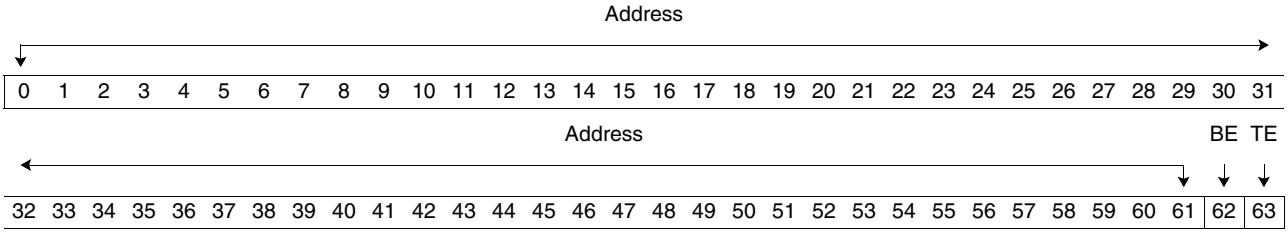
The Instruction Address Breakpoint Register can be used as a debug tool to trigger an event upon the fetch of a particular instruction address. The address in the IABR is compared to the Instruction Fetch Address Register, which will also contain addresses of speculative instruction fetches. The IABR is set up as described in the *PowerPC Microprocessor Family: The Programming Environments* manual, except, in the 970MP microprocessor, the IABR is only available as a trigger to the debug logic. This trigger can be programmed to perform functions such as quiesce or checkstop. If the word specified in the IABR is fetched, the instruction breakpoint handler is invoked. The instruction that triggers the breakpoint does not execute before the handler is invoked.

CIABR can be enabled by either HID0[22] (software accessible) or scan/SCOM override.

The IABR uses the IFU FETCH address, not the current instruction address (CIA) that is executing. An IABR match occurs on the fetch of any instruction, even a speculative instruction.

Note: There can be multiple IABR matches for a single instruction before it is actually executed (or completed).

During power-on reset, all bits are reset to '0'.



Bits	Field Name	Description
0:61	Address	Word address to be compared
62	BE	Breakpoint enabled. An address match causes a trigger to the debug logic.
63	TE	Translation enabled. An IABR match is signaled if this bit matches MSR[IR].

2.1.2.5 Instruction Match CAM Array Access Register (IMC)

The instruction match CAM (IMC) array facility is used for performance monitoring instrumentation. This latter use is restricted for the support processor and is not available through SPR access to this register array. The array has privileged write access and user-level read access through this SPR. Writes to the register array are used to configure the IMC, and reads return information about the availability of registers within the facility. See *Section Instruction Match CAM (IMC) Register* on page 412 for additional details on the IMC register.

2.1.2.6 Performance Monitor Registers (MMCR0, MMCR1, MMCRA, PMC1-8)

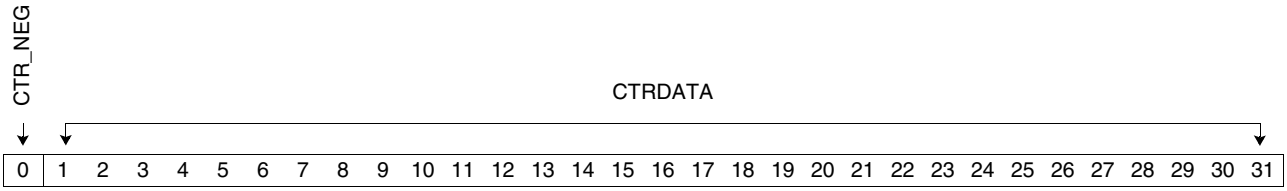
The Performance Monitor Counter Registers (PMC1-PMC8) and the Performance Monitor Control Registers (MMCR0, MMCR1, MMCRA) are supported in the 970MP microprocessor.

The Performance Monitor Control Registers, MMCR0, MMCR1, and MMCRA, are used with the MSR and other SPRs to set up the performance monitor enable states, interrupt conditions, threshold values, match criteria, and selection of the events counted in each of the Performance Monitor Counter Registers, PMC1-PMC8.

The MMCRx Register bit assignments are shown in *Section 10.4.1* on page 301; *Section 10.4.2* on page 304; and *Section 10.4.3* on page 307. All of the MMCRx and PMCx Registers flush to zero unless otherwise noted in the MMCRx and PMCx tables.

The MSR bits that relate to performance monitor functions are shown in *Section 4.3.3* on page 134. The value of the SRR1 Registers when a performance monitor interrupt is taken is shown in *Chapter 10 970MP Performance Monitor*.

Performance Monitor Counter Registers (PMC1-8)

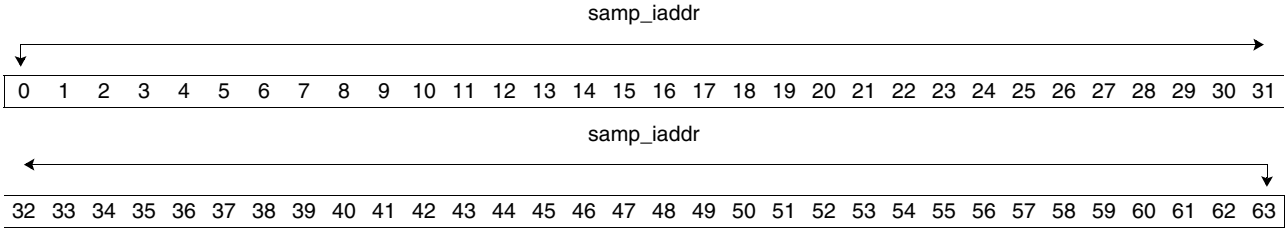


Bits	Field Name	Description
0	CTR_NEG	Counter negative bit
1:31	CTRDATA	Count data

2.1.2.7 Sampled Instruction Address and Sampled Data Address Registers (SIAR, SDAR)

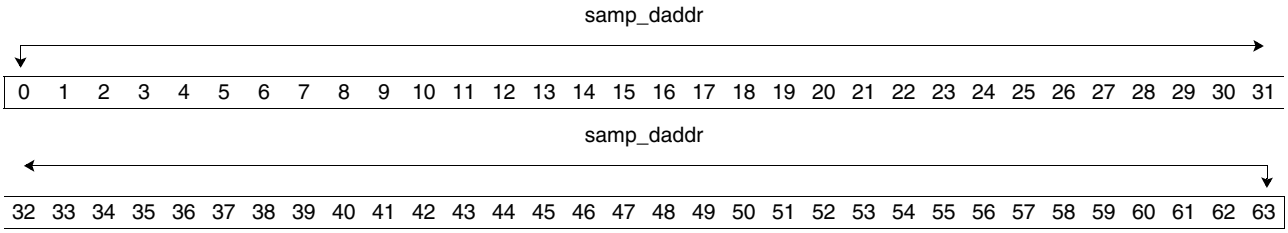
The Sampled Instruction Address Register (SIAR) and the Sampled Data Address Register (SDAR) are supported in the 970MP microprocessor. The SIAR is used to save the effective address of a sampled instruction and the SDAR is used the effective address of a storage operand for a sampled instruction, when the processor is in either trace-marking mode or performance-marking mode. The terms 'sampled' and 'marked' are used interchangeably in this document.

Sampled Instruction Address Register (SIAR)



Bits	Field Name	Description
0:63	samp_iaddr	Sampled Instruction Address

Sampled Data Address Register (SDAR)



Bits	Field Name	Description
0:63	samp_daddr	Sampled Data Address

2.1.2.8 Scan Communication Registers (SCOMC and SCOMD)

Each 970MP processing unit includes a pair of registers to aid in communicating with the Scan Communications facility (SCOM). The SCOMC Register is a control register that includes a command field, a destination field, and a set of status bits. The SCOMD Register is an associated data register that acts as either a source of data or as a destination for data depending on the command placed into the SCOMC Register.

The SCOM facility contains an arbiter, which serializes use of the facility among the bus masters (processor cores and core service processor). However, there are very specific programming conventions associated with the use of this facility. See *Chapter 12 SCOM Interface and Registers on 385* for a detailed description of the SCOM facility.

2.1.2.9 Hypervisor Decrementer Interrupt Register (HDEC)

The Hypervisor Decrementer Interrupt Register (HDEC) is a 32-bit decrementing counter that provides a mechanism for causing a hypervisor decrementer interrupt after a programmable delay.

The HDEC is driven by the same frequency as the Time Base Register, and in the same manner as the Decrementer Register. The Hypervisor Decrementer Register counts down, causing an interrupt and is implemented in SPR 310.

2.1.2.10 Hypervisor Save/Restore Register (HSRR0, HSRR1)

The Hypervisor Machine Status Save/Restore Register 0 (HSRR0) is located in SPR 314 and HSRR1 is located in SPR 315. When a hypervisor decrementer interrupt occurs, the state of the machine is saved in the Hypervisor Machine Status Save/Restore Registers (HSRR0 and HSRR1). The effective address is stored in HSRR0 and the MSR in HSRR1. The contents of these registers is used to restore machine state when a **hrfid** instruction is executed.

2.1.2.11 Hypervisor SPRGs (HSPRG0, HSPRG1)

HSPRG0 and HSPRG1 are 64-bit registers provided for use by hypervisor programs. HSPRG0 is located at SPR 304 and HSPRG1 is located at SPR 305.

Note: Neither the contents of the HSPRGs, nor accessing them using mtspr or mfspr, has a side effect on the operation of the processor. One of more of the registers is likely to be needed by hypervisor interrupt handler programs (for example, as scratch registers, pointers, or both to processor save areas).

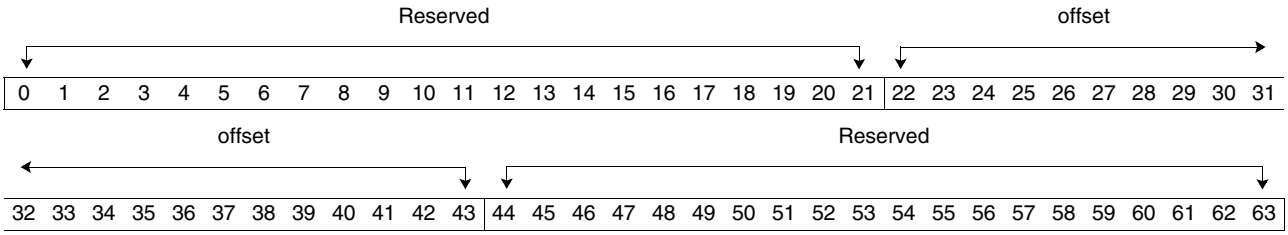
2.1.2.12 Trigger Registers (TRIG0, TRIG1, TRIG2)

Writes to the Trigger Registers, named TRIG0, TRIG1, and TRIG2, can be inserted in the instruction stream to cause triggers to the on-chip debug logic of the trace array. These are intended to be used for lab debug and bring-up only and architecturally behave as a no-op.

2.1.2.13 Hardware Interrupt Offset Register (HIOR)

The Hardware Interrupt Offset Register (HIOR) should be scanned (the HIOR is on the mode ring) to the system's starting address during initialization. Subsequently, the HIOR should be set to zero.

The physical address of the interrupt vector is found using HIOR[22:43] combined with the 20-bit vector offset for the particular exception.



Bits	Field Name	Description
0:21	—	Reserved
22:43	offset	Offset
44:63	—	Reserved

2.2 Instruction Set Summary

This section describes instructions and addressing modes defined for the 970MP microprocessor. These instructions are divided into the following execution unit categories:

- Fixed-Point Processor
- Floating-Point Processor
- Vector Processor
- Load-and-Store Processor
- Branch and Flow Control
- Storage Control
- Memory Synchronization

Fixed-point instructions operate on byte, half-word, word, and double-word operands. Floating-point instructions operate on single-precision and double-precision floating-point operands. The PowerPC Architecture uses instructions that are 4 bytes long and word-aligned. It provides for byte, half-word, word, and double-word operand loads and stores between memory and a set of 32 General-Purpose Registers (GPRs). It provides for word and double-word operand loads and stores between memory and a set of 32 Floating-Point Registers (FPRs). The VPU extension to the PowerPC Architecture provides for quadword operand loads and stores between memory and a set of 32 Vector Registers.

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

2.2.1 Classes of Instructions

The 970MP microprocessor instructions belong to one of the following three classes:

- Defined
- Illegal
- Reserved

Note: While the definitions of these terms are consistent among the PowerPC processors, the assignment of these classifications is not.

The class is determined by examining the primary opcode and the extended opcode, if any. If the opcode or the combination of opcode and extended opcode, is not that of a defined instruction or of a reserved instruction, then the instruction is illegal. Instruction encodings that are now illegal may become assigned to instructions in the architecture or may be reserved by being assigned to processor-specific instructions.

2.2.1.1 Definition of Boundedly Undefined

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

2.2.1.2 Defined Instructions

The 970MP microprocessor provides support for the following optional instructions:

fsqrt	Floating-Point Square Root
fsqrts	Floating-Point Square Root Single
fres	Floating-Point Reciprocal Estimate Single
frsqrte	Floating-Point Reciprocal Square Root Estimate A-Form
fsel	Floating-Point Select
mfsr	Move from Segment Register
mfsrin	Move from Segment Register Indirect
mtmsr	Move to Machine State Register (32-bit)
mtsr	Move to Segment Register
mtsrin	Move to Segment Register Indirect
slbie	SLB Invalidate Entry
slbia	SLB Invalidate All
tlbie	TLB Invalidate Entry
tlbsync	TLB Synchronize

The 970MP microprocessor does *not* provide support for the following optional or obsolete instructions (or instruction forms). Attempted use of these will result in an illegal instruction type of program interrupt.

bccbr	Branch Conditional to CBR (obsolete)
dcba	Data Cache Block Allocate (obsolete)
dcbi	Data Cache Block Invalidate (obsolete)
eciwx	External Control In Word Indexed
ecowx	External Control Out Word Indexed
mcrxr	Move to Condition Register from XER Register (obsolete)
mtsrd	Move to Segment Register Double Word (obsolete)
mtsrdin	Move to Segment Register Double Word Indirect (obsolete)
rfi	Return from Interrupt (obsolete)
tlbia	TLB Invalidate All
tlbiex	TLB Invalidate Entry by Index (obsolete)
slbiex	SLB Invalidate Entry by Index (obsolete)

2.2.1.3 Invalid Forms

In general, the 970MP microprocessor handles invalid forms of instructions in the manner that is most convenient for the particular case. This document specifies the cases where a system-level error handler will be invoked, but does not always describe actions for other cases of invalid forms. IBM does not recommend that software or other system facilities use the 970MP microprocessor behavior in these cases, because it is not formally specified and may change as the design matures.

The 970MP processing unit ignores the state of reserved bits in the instructions (denoted by "/, //, or / / /" in the instruction definition) and executes the instruction normally. Only results that differ from those specified by the architecture are described in *Table 2-5*.

Table 2-5. Invalid Forms of Instructions

Instruction	Form	970MP Action
Any	Bits in reserved fields.	Bits in reserved fields are ignored; the results of executing an instruction in which one or more reserved bits is '1' is the same as if the bits were '0'.
Store Word Conditional Indexed (**stwcx**) or Store Double Word Conditional Indexed (**stdcx**)	EA not equal to the EA of the previous **lwarx** or **ldarx**	Whether the store will be performed is indeterminate, but the setting of CR[0] will correctly reflect the outcome of the store.
stwcx or **stdcx**	Storage operand is not aligned.	If the instruction specifies an effective address that is not a multiple of the operand length, it causes an alignment interrupt to be invoked.
stwcx or **stdcx**	Storage exception and reservation nonexistent	The instruction causes a data storage interrupt.
Move From Time Base (**mftb**)		The instruction produces the same result as the **mfspr** instruction.
mtspr and **mfspr**		For a complete description of the **mtspr** and **mfspr** invalid forms, see *Section 2.1.2.1 Move To and Move From System Register Instructions* on page 52.

2.2.1.4 Illegal Instructions

Illegal instructions can be grouped into the following categories:

- Instructions not defined in the PowerPC Architecture. The following primary opcodes are defined as illegal, but may be used in future extensions to the architecture: 1, 5, 6, 56, 57, 60, 61.

  Note: Primary opcode 4 is used in the 970MP microprocessor to implement the vector extensions that are described in *Chapter 10 970MP Performance Monitor*.

- Instructions defined in the PowerPC Architecture but not implemented in a specific PowerPC implementation. For example, the following primary opcodes that are legal on 64-bit PowerPC processors are considered illegal by 32-bit processors: 30, 62.

  Note: On the 970MP microprocessor, these instructions are executed in 32-bit mode.

- All unused extended opcodes for instructions. Notice that extended opcodes for instructions defined only for 64-bit implementations are illegal in 32-bit implementations. The following primary opcodes have unused extended opcodes: 19, 30, 31, 56, 57, 59, 60, 61, 62, 63. (Primary opcodes 30 and 62 are illegal for 32-bit implementations, but as 64-bit opcodes they have some unused extended opcodes.)

- An instruction consisting entirely of zeros is guaranteed to be an illegal instruction. This increases the probability that an attempt to execute data or uninitialized memory invokes the system illegal instruction error handler (a program exception).

See *Section 4.5.9 Program Exception* on page 142 for additional information about illegal and invalid instruction exceptions. Except for an instruction consisting of binary zeros, illegal instructions are available for additions to the PowerPC Architecture.

2.2.1.5 Reserved Instructions

The PowerPC Architecture breaks the reserved instruction class down into several categories. The 970MP microprocessor behaves as described below for each category of reserved instructions:

- Primary opcode equals zero. The 970MP processing unit will take an illegal instruction type of program interrupt for all cases except the Support Processor Attention (**attn**) instruction when HID0[31] is set to '1'.
- Power Architecture™ instructions not in the PowerPC AS Architecture. The 970MP processing unit will take an illegal instruction type of program interrupt.
- Implementation-specific instructions used to conform to the architecture. No action taken.
- Other instructions. The 970MP processing unit supports the implementation-specific instruction, **tlbiel** (the processor local form of the TLB Invalidate entry used for managing TLB parity errors).

In addition, several implementation-specific registers are available for access through the **mtspr** and **mfspr** instructions. These are described in *Section 2.1.2.1 Move To and Move From System Register Instructions* on page 52.

2.2.2 Instruction Set Overview

The following sections provide a brief overview of the PowerPC instructions implemented in the 970MP microprocessor and highlight how a 970MP microprocessor implements a particular instruction. Note that the categories used in this section correspond to those used in *"Addressing Modes and Instruction Set Summary"* in the *PowerPC Microprocessor Family: The Programming Environments* manual. These categorizations are somewhat arbitrary and are provided for the convenience of the programmer. They do not necessarily reflect the PowerPC Architecture specification.

Note: Some instructions have the following optional features:

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

2.2.3 Fixed-Point Processor

2.2.3.1 Fixed-Point Arithmetic and Compare Instructions

The architecture states that instructions that have the overflow exception (OE) bit set, or instructions that can set the carry (CA) bit, may execute more slowly than instructions that do not. In the 970MP microprocessor, the summary overflow (SO) bit in the XER is not renamed. For instructions with the OE set, it is initially assumed that no overflow will occur and that the SO bit does not need to be changed. If the instruction does cause an overflow and the SO bit was not set before the instruction executed (and therefore needs to be set),

the machine will flush this instruction and those beyond this instruction, set the non-renamed SO bit, and then refetch and re-execute the instructions that follow. In general, if no overflow occurs or the SO bit has already been set, this strategy will not have an adverse effect on performance.

Alternatively most instructions that set and use the CA bit do not have any particular performance considerations. This field of the XER is renamed, and many of the common dependence hazards are minimized. For more information about the performance implications associated with the use of XER bits, see *Section 6.3.5.5 XER Renaming* on page 185.

2.2.3.2 Fixed-Point Logical, Rotate, and Shift Instructions

The architecture defines the preferred no-op to be OR Immediate (**ori**) 0,0,0. In the 970MP microprocessor, this no-op form is recognized by the hardware and allowed to complete without taking any execution resources. This makes the instruction valuable for padding other instructions to achieve better alignment or better separation.

2.2.3.3 Move to and Move from System Register Instructions

The **mtspr** instruction provide access to system registers using a GPR as the source register. The **mfspr** instruction provides access to the system registers using a GPR as the destination register. *Table 2-3 Implementation-Specific SPRs* on page 53 lists the SPR numbers for both user-level and supervisor-level access to 970MP-specific registers.

2.2.3.4 Move to and Move from Machine State Register

The 970MP microprocessor supports both the 32-bit **mtmsr** instruction and the 64-bit **mtmsrd** instruction. The 970MP microprocessor works to optimize the **mtmsr** instruction to help speed up cases where little or no synchronization is required (that is, updates to the MSR[EE] bit).

2.2.3.5 Fixed-Point Invalid Forms and Undefined Conditions

The results of executing an invalid form of a fixed-point instruction or an instance of a fixed-point instruction for which the architecture specifies that some results are undefined are described below for the cases in which executing an instruction does not cause an exception. Only results that differ from those specified by the architecture are described in the following list.

- **Instruction with Reserved Fields**
  Bits in reserved fields are ignored; the results of executing an instruction in which one or more reserved bits are '1' is the same as if the bits were '0'.
- **Divide Word (divw), Divide Word Overflow (divwo), Divide Word Unsigned (divwu), and Divide Word Unsigned Overflow (divwuo) Instructions**
  Bits 0:31 of the **r**D[1] are set to x'0000 0000'.
- **Multiply High Word (mulhw) and Multiply High Word Unsigned (mulhwu) Instructions**
  rD bits 0:31 contain the same result as **r**D[32:63].
- **Divide Instructions (divide by zero)**
  If the divisor is zero, **r**D is set to zero. If, in addition, the record bit (RC) in the vector instruction field equals '1', CR0 is set to '0010'.
- **Move To and Move From Special Purpose Register Instructions**
  Table 2-4 on page 55 describes the results of specifying an SPR value that is not defined for the implementation.
- **Move From Time-Base Instruction**
  The **mftb** instruction is treated as an alias for the **mfspr** instruction; the results are the same as for executing an **mfspr** instruction.
- **Move From Condition Register Instruction** (bit 11 is set to '1')
  One CR field is copied into the associated bits of the **r**D, and the remaining bits of the **r**D are set to zeros.
- **Move From Condition Register Instruction** (bit 11 is set to '1', and multiple bits of the FXM[2] field are set to '1')
  The source is CR(**n**), where **n** is the CR field specified by the bit in the FXM that is set and has the smallest index value. If no bit in FXM is set to '1', the results will be the same as if the FXM was set to '00000001'.
- **Move To Condition Register Fields Instruction** (bit 11 is set to '1', and multiple bits of the FXM field are set to '1')
  CR(**n**) is updated where **n** is the CR field specified by the bit in FXM that is set and has the smallest index value. If no bit in the FXM is set to '1', executing the instruction does not modify the CR.

1. Field used to specify a General Purpose Register (GPR) to be used as a target.
2. Field mask used to identify the CR fields to be updated by the **mtcrf** instructions.

2.2.4 Floating-Point Processor

Each 970MP processing unit contains two double-precision floating-point units. Each of these units is optimized for fully pipelined double-precision multiply-add functionality. In addition, each unit is capable of performing floating-point divide and square root instructions. For more information about the performance of floating-point operations, see *Section 6.3 Performance Features, Mechanisms, and Hazards* on page 160.

The optional floating-point instructions (**fsqrt**, **fsqrts**, **fres**, **frsqrte**, and **fsel)** defined in the *PowerPC Microprocessor Family: The Programming Environments* manual are implemented.

Note: The 970MP microprocessor does *not* support the non-IEEE mode that was defined in earlier versions of the architecture.

2.2.4.1 Floating-Point Arithmetic Instructions

The architecture requires operands for single-precision floating-point arithmetic instructions to be representable in single-precision format. If they are not, then the results of the single-precision arithmetic instructions are undefined. For the single-precision divide and square-root instructions, **fdivs** and **fsqrts**, single-precision algorithms are executed on the double-precision data with the final results rounded to single-precision.

2.2.4.2 Floating-Point Invalid Forms and Undefined Conditions

The results of executing an invalid form of a floating-point instruction or an instance of a floating-point instruction for which the architecture specifies that some results are undefined are described below for the cases in which executing an instruction does not cause an exception. Only results that differ from those specified by the architecture are described.

- **Instruction with Reserved Fields**
  Bits in reserved fields are ignored; the results of executing an instruction in which one or more reserved bits are '1' is the same as if the bits were '0'.
- **Floating-Point Convert to Integer Word Instructions: fctiw** or **fctiwz**
  The Instruction target register (**fr**D[0:31]) is set to x'FFF8 0000'.
- **Floating-Point Convert to Fixed-Point Instructions** (**fctiw, fctiwz, fctid,** and **fctidz**)
  The contents of FPSCR(FPRF) are set to '00000'.
- **Move from FPSCR Instruction**
  frD[0:31] is set to x'FFF8 0000'.

2.2.5 Vector Processor

Each 970MP processing unit contains two vector units: the vector arithmetic logical unit (VALU) and the vector permute unit (VPERM). The vector instructions and their implementation are described in *Chapter 13 Vector Processing Unit*.

2.2.6 Load Store Processor

2.2.6.1 Floating-Point Load-and-Store Instructions

Most forms of unaligned floating-point storage accesses are executed entirely in hardware (see *Section 3.3.2.1 Storage Access Alignment Support* on page 87).

2.2.6.2 Fixed-Point Load Instructions

Most forms of unaligned load operations are executed entirely in hardware. If a basic load operation crosses a page boundary, and either page translation signals an exception condition, then when the interrupt occurs it will appear as though none of the load instructions have executed. This is not always the case for load multiple or load string instructions. For more information, see *Section 2.2.6.4 Fixed-Point Load-and-Store Multiple Instructions* and *Section 2.2.6.5 Fixed-Point Load-and-Store String Instructions* on page 77.

The Load Algebraic, Load with Byte Reversal, and Load with Update instructions may have greater latency than other load instructions. These instructions are implemented as a sequence of internal operations. Due to the dynamic scheduling and out-of-order execution capability of the processor, these effects are somewhat minimized. It should also be noted that, although these instructions are broken up in this manner, the effects are never visible from a programming model perspective. For more information about the performance of these instructions, see *Section 6.4.1 Instruction Latencies and Throughputs* on page 203.

Any load from storage marked cache-inhibited that is not aligned will cause an alignment interrupt.

2.2.6.3 Fixed-Point Store Instructions

Most forms of unaligned store operations are executed entirely in hardware. If a store operation crosses a page boundary, and the second page translation signals an exception condition, then after the interrupt is taken it will appear as though none of the storage updates have occurred to either page. (This is not always the case for store multiple or store string instructions. See *Section 2.2.6.4 Fixed-Point Load-and-Store Multiple Instructions* and *Section 2.2.6.5 Fixed-Point Load-and-Store String Instructions* on page 77 for more information.)

Any store to storage marked cache-inhibited that is not aligned will cause an alignment interrupt.

2.2.6.4 Fixed-Point Load-and-Store Multiple Instructions

The Load Multiple Word (**lmw**) instruction is executed so that up to two registers are loaded each cycle. Similarly, the Store Multiple Word (**stmw**) instruction is executed so that up to two registers are stored each cycle. The 32-entry store queue can accept up to two 8-byte stores per cycle; the cache can accept one 8-byte store per cycle. Because these instructions are emulated through the use of microcoded templates (see *Section 6.3.3.2 Microcode Template-Based Instruction Cracking* on page 175 for more information), after a small start-up penalty, they are processed at a rate of up to two registers per cycle.

Most forms of **lmw** and **stmw** instructions, even those that cross page and segment boundaries, are executed entirely in hardware. These instructions and the individual storage accesses associated with the instructions are not atomic. If an **stmw** crosses a page boundary, and the second page translation signals an exception condition, then after the interrupt is taken it will appear as though none, some, or all of the accesses to the first page have occurred. It will also appear as though none of the accesses to the second page have occurred. However, for an **lmw** instruction that crosses a page boundary where the second page translation signals an exception condition, all of the target registers will have an undefined value.

An attempt to execute a non-word aligned **lmw** or **stmw** will cause an alignment interrupt. An attempt to execute an **lmw** or **stmw** to storage marked cache-inhibited will also cause an alignment interrupt.

The architecture allows these instructions to be interrupted by certain types of asynchronous interrupts (external interrupts, decrementer interrupts, machine check interrupts, and system reset interrupts). In these cases, for the load multiple instructions, all of the registers that were to be updated will have an undefined value. The instruction must be completely restarted to achieve the full effect (that is, no partial restart capability is supported). For the store multiple instructions, some of the storage locations referred to by the instruction may have been updated. However, to guarantee full completion of the store multiple instructions, they must also be completely restarted.

2.2.6.5 Fixed-Point Load-and-Store String Instructions

The Load String Word (**lsw**) instruction is executed so that up to two registers are loaded each cycle. Similarly, the Store String Word (**stsw**) instruction is executed so that up to two registers are stored each cycle. The 32-entry store queue can accept up to two 8-byte stores per cycle; the cache itself can only accept one 8-byte store per cycle.

Because the immediate forms of these instructions are implemented using microcoded templates (see *Section 6.3.3.2 Microcode Template-Based Instruction Cracking* on page 175 for more information) they incur a small start-up penalty. The X-form of the instructions contains a dependency on bits in the fixed-point XER Register. Therefore, depending on when the last update to these bits occurred, the instruction may be subject to a more expensive runtime flush and emulate sequence. For more information about the performance of these instructions, see *Section 6.4.1 Instruction Latencies and Throughputs* on page 203.

Most Load String and Store String instructions that cross page or segment boundaries are executed entirely in hardware. If a Store String crosses a page boundary, and the second page translation signals an exception condition, then after the interrupt is taken it will appear as though none, some, or all of the accesses to the first page have completed. It will also appear as though none of the accesses to the second page have occurred. However, for Load String instructions that cross a page boundary where the second page translation signals an exception condition, all of the target registers will have an undefined value.

If the storage operand of a Load String Word Immediate (**lswi**) instruction is word aligned, then the accesses are performed in an optimal manner. If the operands are so aligned, the accesses are performed in an optimal manner if the operand resides entirely within a 64-byte block that is resident in the L1 D-cache or resides entirely within a 32-byte block. Although other unaligned string operations are supported in hardware, they may cause machine flushes and require long sequences of microcode. As a result, these types of unaligned string instructions may have significantly longer latencies.

An attempt to execute an **lswi**, Load String Word Indexed (**lswx**), Store Sting Word Immediate (**stswi**), or Store String Word Indexed (**stswx**) instruction to storage marked cache-inhibited will cause an alignment interrupt.

The architecture allows these instructions to be interrupted by certain types of asynchronous interrupts (external interrupts, decrementer interrupts, machine check interrupts, and system reset interrupts). In these cases, for the Load String instructions, all of the registers that were to be updated will have an undefined value. The instruction must be completely restarted to achieve the full effect (that is, no partial restart capability is supported). For the Store String instructions, some of the storage locations referred to by the instruction may have been updated.

The architecture describes some preferred forms for the use of load-and-store string instructions. In the 970MP microprocessor, these preferred forms have no effect on the performance of the instructions.

2.2.6.6 Load/Store Invalid Forms and Undefined Conditions

The results of executing an invalid form of a load/store instruction for which the architecture specifies that some results are undefined are described below for the cases in which executing an instruction does not cause an exception. Only results that differ from those specified by the architecture are described.

- **Load with Update Instructions** (**r**A[1] is set to '0')
  The storage operand addressed by the EA is placed into **r**D. The sum of that storage operand and **r**B is placed in **r**A.
- **Load with Update Instructions** (**r**A equals **r**D)
  The EA is placed into the **r**D. The storage operand addressed by the EA is accessed, but the data returned by the load is discarded.
- **Load Multiple Instructions** (**r**A is in the range of registers to be loaded)
  If an exception (for example, a data storage or external exception) causes the execution of the instruction to be interrupted, the instruction is restarted, the **r**A has been altered by the previous partial execution of the instruction, and the **r**A does not equal '0', the new contents of the **r**A are used to compute the EA.
- **Load Multiple Instructions** (causing a misaligned access)
  For a Load Multiple Word instruction, if the storage operand specified by the EA is not a multiple of four, an alignment exception is taken. For a Load Multiple Double Word instruction, if the storage operand specified by the EA is not a multiple of eight, an alignment exception is taken.
- **Load String Instructions** (zero length string)
  The **r**D is not altered.
- **Load String Instructions** (**r**A, or **r**B[2], or both are in the range of registers to be loaded)
  If **r**A, or **r**B, or both are in the range of registers to be loaded, the results are as follows:

  Indexed Form: If **r**A is set to '0', let Rx be **r**B; otherwise let Rx be the register specified by the smaller of the two values in instruction fields **r**A and **r**B. If the **r**D equals Rx, no registers are loaded. Otherwise, registers **r**D through RX-1 are loaded as specified in the architecture (that is, only part of the storage operand is loaded).

  Immediate Form: If **r**A is set to '0', the instruction is executed as if it were a valid form. If **r**A equals **r**D, no registers are loaded; otherwise, registers **r**D through **r**A-1 are loaded as if the instruction was a valid form but specifying a shorter operand length**.**
- **Store with Update Instructions** (**r**A is set to '0')
  EA is placed into R0.
- **Load or Store Floating-Point with Update Instructions** (**r**A is set to '0')
  EA is placed into R0.
- **Floating-Point Store Single Instructions** (exponent less than 874 and FRS[09:31] not equal to '0')
  The value placed in storage is a '0' with the same sign as the value in the register.

1. Field used to specify a GPR to be used as a source or as a target.
2. Field used to specify a GPR to be used as a source.

2.2.7 Branch Processor

2.2.7.1 Branch Processor Instructions

Support Processor Attention Instruction

The 970MP microprocessor supports a special, implementation-dependent instruction for signalling an attention to the support processor.

Figure 2-2. Processor Attention Instruction

attn

00	I	256	/
0	6	21	31

The immediate field (**I**) has no effect on the operation of this instruction in the 970MP microprocessor. If the support processor attention enable bit is set (HID0[31] = '1'), this instruction will cause all preceding instructions to run to completion, the machine to quiesce, and the assertion of the support processor attention signal. If the support processor attention enable is not set (HID0[31] = '0'), this instruction will cause an illegal instruction type of program interrupt.

2.2.7.2 Branch Processor Instructions with Undefined Results

The results of executing an invalid form of a branch instruction or an instance of a branch instruction for which the architecture specifies that some results are undefined are described below. Only results that differ from those specified by the architecture are described.

- **Instructions with Reserved Fields**
  Bits in reserved fields are ignored. The results of executing an instruction in which one or more of these bits is '1' is the same as if the bits were '0'.
- **bcctr and bcctrl Instructions**
  If branch-options (BO)[2] is set to '0', the contents of the CTR before any update are used as the target address and to test the contents of the CTR to resolve the branch. The contents of the CTR are then decremented and written back to the CTR.
- **System Call (sc) Instructions** (opcode 17)
  Bits 30:31 Description
  '00' **sc** instruction
  '01' illegal instruction exception
  '10' **sc** instruction
  '11' **sc** instruction

2.2.7.3 Move To Condition Register Fields Instruction

The architecture warns that updating a subset of the CR fields on a Move to Condition Register Fields (**mtcrf**) instruction may have worse performance than updating all of the fields. In the 970MP microprocessor, both the **mtcrf** instruction and the Move from Condition Register (**mfcr**) instruction are emulated through the microcode templates. For best performance, software should use the new, single-field variants of these instructions as described in the architecture. For more information about the performance of these instructions, see *Section 6.3 Performance Features, Mechanisms, and Hazards* on page 160.

The 970MP microprocessor supports the optional architecture extension that defines slight variations to the **mtcrf** and **mfcr** instructions to indicate that the movement of a single field of the Condition Register is desired. Because the performance of these instructions is better than their multiple field counterparts, use of these instructions is encouraged.

2.2.8 Storage Control Instructions

2.2.8.1 Key Aspects of Storage Control Instructions

In each 970MP processing unit, all cache control instructions operate on aligned 128-byte sections of storage. *Table 2-6* summarizes many of the key aspects of the storage control instructions.

Table 2-6. Storage Control Instructions

Aspect	Cache Instructions					
	icbi	**dcbt**	**dcbtst**	**dcbz**	**dcbst**	**dcbf**
Granularity	128 bytes	128 bytes	128 bytes	32 or 128 bytes	128 bytes	128 bytes
Semantic checking	Load (DSI on storage exception)	Load (no-op on storage exception)	Load (no-op on storage exception)	Store (DSI on storage exception)	Load (DSI on storage exception)	Load (DSI on storage exception)
"r" bit update	Yes	Yes	Yes	Yes	Yes	Yes
"c" bit update	No	No	No	Yes	No	No
L1 I-cache effect	L1 I-cache and prefetch buffer	None	None	None	None	None
L1 D-cache effect	None	See *Section 2.2.8.4*	See *Section 2.2.8.4*	Invalidate	No-op	Invalidate
L2 Cache effect	None	See *Section 2.2.8.4*	See *Section 2.2.8.4*	See *Section 2.2.8.5*	See *Section 2.2.8.6*	See *Section 2.2.8.7*
TLB effect	Reload as required	Reload as required	Reload as required	Reload as required	Reload as required	Reload as required
SLB effect	Reload as required	None (no-op if miss)	None (no-op is miss)	Reload as required	Reload as required	Reload as required

2.2.8.2 Instruction Cache Block Invalidate (icbi)

The instruction cache block size for **icbi** on the 970MP processing unit is 128 bytes.

Execution of this instruction occurs in multiple phases. First, the effective address is computed and translated by the load/store execution pipeline. Next, the resulting real address is passed to the 970MP STS logic which broadcasts it onto the system bus. When the 970MP STS snoops this type of command on the system bus, it presents the command to the upstream instruction caches. As these invalidates are presented to the instruction cache, the associated real addresses are checked against all 16 possible locations in the effective-addressed I-cache that could contain the particular real address. Only entries that actually match the real address will be invalidated. In addition, all entries in the instruction prefetch queue will be invalidated (independent of the address). As an aid for quickly flushing the entire contents of the I-cache, a special mode bit is provided (HID1[9]) that forces each of these 16 entries to be invalidated on an **icbi** (even if their address does not match the invalidate address). For more information about this instruction, see *Section 6.3.1.5 ICBI Instruction Handling* on page 163.

The **icbi** instruction has no effect on the L2 cache.

To ensure that the storage access caused by an **icbi** instruction has been performed with respect to the processor executing the **icbi** instruction, an **isync** instruction must be executed on that processor.

2.2.8.3 Instruction Cache Synchronize (isync)

As a performance optimization, the 970MP microprocessor internally tracks and updates a scoreboard bit for instructions that change instruction-cache-oriented context that are required to be synchronized by the **isync** instruction. When the **isync** instruction is executed, this scoreboard bit is checked to see whether the machine must flush and refetch the instructions following the **isync**. In addition, the **isync** instruction is often used as a load barrier to prevent any subsequent load (or store) instructions from executing before previous load instructions have been completed. In these cases, the scoreboard bit will typically not be set, and **isync** can complete without causing a flush.

2.2.8.4 Data Cache Block Touch (dcbt and dcbtst)

The data cache block size for **dcbt** and **dcbtst** on the 970MP processing unit is 128 bytes.

These instructions act as a touch for the D-cache hierarchy and the TLB. If data translation is enabled (MSR[DR] is set to '1'), and an SLB miss results, then the instruction will be treated as a no-op. If a TLB miss results, then the instruction will reload the TLB (and set the reference bit). Once past translation, if the page protection attributes prohibit access, or the page is marked cache-inhibited, or the page is marked guarded, or the processors' D-cache is disabled (using the bits in the HID4 Register), then the instruction will be finished as a no-op and will not reload the cache. Otherwise, the instruction will check the state of the L1 D-cache, and, if the block is not present, it will then initiate a reload. Note that this may also reload the L2 cache with the addressed block if it is not already present. If the cache block is already present in the L1 D-cache, the cache content is not altered. Note that if the **dcbt** or **dcbtst** instruction does reload cache blocks, it will affect the state of the cache replacement algorithm bits.

The 970MP microprocessor does implement the optional extension to the **dcbt** instruction that allows software to directly engage a data stream prefetch from a particular address. For more information about data stream prefetch, see *Section 3.5.4 Data Prefetch* on page 110.

2.2.8.5 Data Cache Block Zero (dcbz)

The data cache block size for **dcbz** on the 970MP processing unit is 128 bytes. Support is also provided for a **dcbz** of 32 bytes in order to accommodate coding that assumes a 32-byte block size. The **dcbz** actions are listed in *Table 2-7*.

Note: The entire instruction cache must be flushed whenever HID5[56] or HID5[57] are changed.

*Table 2-7. **dcbz** Actions*

HID5[57]	HID5[56]	**dcbz** Instruction Bit 10	Action
1	X	0	Illegal instruction
X	X	1	Cache block (128 bytes) zeroed
0	0	0	Cache block (128 bytes) zeroed
0	1	0	32-byte block zeroed

The function of **dcbz** is performed in the L2 cache. As a result, if the block addressed by the **dcbz** is present in the L1 D-cache, then the block will be invalidated before the operation is sent to the L2 cache logic for execution. The L2 cache will gain exclusive access to the block (without actually reading the old data), and will perform the zeroing function. For the 32-byte **dcbz**, the L2 cache may be required to read the line and then zero the 32 bytes.

If the cache block specified by the **dcbz** instruction contains an error, even one that is not correctable with error checking and correction (ECC), the contents of all locations within the block are set to zeros in the L2 cache. If the specified block in the L2 cache does not contain a hard fault, a subsequent load from any location within the cache block will return zeros and not cause a machine check interrupt.

If the block addressed by the **dcbz** instruction is in a memory region marked cache-inhibited, or if the L1 D-cache or L2 cache is disabled (using the bits in HID registers), then the instruction will cause an alignment interrupt to occur.

Implementation Note: In order to emulate the behavior of the obsolete **dcba** instruction, a mode bit is provided that changes the behavior of **dcbz** as follows. When the mode bit is set to '1', if the block addressed by the **dcbz** instruction is in a memory region marked cache-inhibited, the instruction is treated as a no-op. The **dcba** instruction was defined such that the referenced and changed bits need not be updated in this case. However, the 970MP microprocessor will update these bits.

2.2.8.6 Data Cache Block Store (dcbst)

The data cache block size for **dcbst** on the 970MP processing unit is 128 bytes.

The **dcbst** instruction forces all preceding stores to the referenced block to become committed to the cache hierarchy, and then forces a clean operation in the L2 cache.

The **dcbst** instruction has no direct effect on the L1 D-cache (because it is store-through, it never contains modified data). The L2 cache updates and processor interconnect bus operations are performed as shown in *Table 3-5 970MP L2 Cache State Transitions Due to Processor Instructions* on page 95 and *Table 3-6 970MP L2 Cache State Transitions Due to Bus Operations* on page 96.

2.2.8.7 Data Cache Block Flush (dcbf)

The data cache block size for **dcbf** on the 970MP processing unit is 128 bytes.

The **dcbf** instruction forces all preceding stores to the referenced block to become committed to the cache hierarchy. It then acts like an invalidate to the L1 D-cache (because it is store-through, it never contains modified data). The L2 cache updates and processor interconnect bus operations are performed as shown in *Table 3-5 970MP L2 Cache State Transitions Due to Processor Instructions* on page 95 and *Table 3-6 970MP L2 Cache State Transitions Due to Bus Operations* on page 96.

2.2.8.8 Load and Reserve and Store Conditional Instructions (lwarx/ldarx, stwcx/stdcx)

The reservation granularity for the 970MP processing unit is 128 bytes.

The **lwarx** and **ldarx** instructions are sometimes executed speculatively. To achieve optimal performance using these instructions, see *Section 6.3 Performance Features, Mechanisms, and Hazards* on page 160 for usage guidelines.

An attempt to execute a non-word aligned **lwarx** or **stwcx**, or a non-double-word aligned **ldarx** or **stdcx** will cause an alignment interrupt. An attempt to execute an **lwarx, ldarx, stwcx,** or **stdcx** instruction to storage marked cache-inhibited will cause a data storage interrupt.

2.2.9 Memory Synchronization Instructions

The 970MP design achieves high performance by exploiting speculative out-of-order instruction execution. The **sync** instruction, as defined in the architecture, acts as a serious barrier to this type of aggressive execution and therefore can have a dramatic effect on performance. Although the 970MP microprocessor has optimized the performance of **sync** to some degree, care should be exercised in the use of this instruction (see *Section 6.4.1 Instruction Latencies and Throughputs* on page 203 for more information about the performance aspects of **sync**). As a performance consideration, software should attempt to use the lightweight version of **sync** (**lwsync**) whenever possible.

The 970MP microprocessor also supports the architected Page Table Entry Synchronization (**ptesync**) instruction for use in synchronizing page table updates. The 970MP microprocessor implements the Enforce In-Order Execution of I/O (**eieio**) instruction as described in the *PowerPC Virtual Environment Architecture (Book II)*.

In the 970MP storage subsystem logic, the store queues above the L2 cache attempt to gather both cacheable and cache-inhibited store operations sequentially to improve bandwidth. A mode bit exists in the BIU Mode Register (at SCOM address x'043000') to disable store gathering of cache-inhibited stores. Alternatively, if store gathering is not wanted, software must insert between successive stores either an **eieio** (preferable for performance) or a **sync** to prevent it. The **eieio** instruction is broadcast onto the system bus to allow ordering to be properly enforced throughout the cache hierarchy and memory system (when detected on the system bus, these transactions have no direct effect on the processor).

2.2.10 Recommended Simplified Mnemonics

To simplify assembly language coding, a set of alternative mnemonics is provided for some frequently used operations (such as no-op, load immediate, load address, move register, and complement register). Programs written to be portable across the various assemblers for the PowerPC Architecture should not assume the existence of mnemonics not described in this document.

For a complete list of simplified mnemonics, see the *PowerPC Architecture books*.

3. Storage Subsystem

The storage subsystem (STS) of the 970MP processing unit encompasses the core interface unit (CIU), the non-cacheable unit (NCU), the L2 cache control unit and the L2 cache, and the bus interface unit (BIU).

This section provides an overview and a high-level block diagram of the storage subsystem. It summarizes key design fundamentals and the storage hierarchy. The functional units are described in detail.

The following key features are fundamental to the design:

- Store-through L1 data cache (D-cache)
- No castouts or snoop pushes by the core
- Non-blocking snoop invalidates to the core (both instruction and data invalidates)
- Integrated L2 controller
- L2 controller handling of cacheable instruction fetches, loads and stores, and **dcbz** instructions.
- Non-cacheable unit handling of other storage type instructions.

3.1 Storage Hierarchy

Table 3-1. Storage Hierarchy Characteristics

Characteristic	L1 Instruction Cache	L1 Data Cache	L2 Cache
Data type	Instructions only	Data only	Instructions and data
Size	64 KB	32 KB	1 MB
Associativity (replacement policy)	Direct map	2-way (least recently used [LRU])	8-way set associate (LRU)
Line size (sector)	128 bytes (4 × 32 bytes)	128 bytes	128 bytes
Operation granularity	128 bytes	128 bytes	128 bytes
Index	Effective address	Effective address	Physical address
Tags	Physical address	Physical address	Physical address
Number of ports	1 read or 1 write (directory has 2 reads or 1 write)	2 reads and 1 write	1 read or 1 write
Inclusiveness	N/A	N/A	Inclusive of L1 D-cache; Not inclusive of L1 instruction cache (I-cache).
Hardware coherency	No	Yes	Yes; separate snoop ports
Store policy	N/A	Store-through; no allocate on store miss	Store back; allocate on store miss
Enable bit	Yes	Yes	No
Reliability, availability, service-ability (RAS)	Parity with invalidate on error for data and tags	Parity with invalidate on error for data and tags	Error checking and correction (ECC) on data; parity on tags (recoverable with redundant tags)

3.2 Caches

The 970MP processing unit contains two levels of cache hierarchy: L1 and L2. The coherence block size for the 970MP processing unit is 128 bytes. For more information about cache characteristics of the 970MP processor, see *Table 3-1 Storage Hierarchy Characteristics* on page 85. For more information about the relative performance of these caches, see *Section 6.4.3.1 Load Latencies* on page 217.

The 970MP processing unit automatically maintains the coherency of all data cached in these caches. Because some levels of the cache hierarchy contain both instructions and data, when the L2 cache services an instruction cache reload request, it does this in a coherent manner. This avoids the scenario where a line is reloaded into the L2 cache on behalf of a non-coherent instruction fetch, but then accessed by a load or store instruction with an aliased address that calls for correct coherency. However, the processor does not maintain instruction storage consistent with data storage and, as described in PowerPC AS Architecture, synchronization code is required to make the two consistent.

The L1 I-cache is indexed with an effective address. As a result, multiple copies of a particular physical block of memory can reside in multiple positions in the L1 I-cache (up to sixteen because four bits of the effective address are used in indexing the cache). The tag comparison associated with lookups in this cache is done using real addresses, so there are no 'synonym' or 'alias' hazards that must be explicitly handled by the system software. For more information about the performance aspects of the L1 I-cache, see *Section 6.3.1 L1 Instruction Cache and Prefetching* on page 160.

The L1 D-cache is indexed with an effective address. Only one copy of a particular real address block is kept in the cache at a time. On each access, a tag comparison is done with the real address. On a cache miss, the cache reload mechanism searches the other three related sets to determine if the required real address block is located elsewhere in the cache. If so it will appropriately eliminate these copies. For more information about the performance aspects of the L1 D-cache, see *Section 6.3.7 L1 Data Cache Management* on page 187.

In addition to maintaining caches, each 970MP processing unit also includes several types of queues that act as logical predecessors and extensions to the caches. In particular, the machine contains store queues for holding store data "above the caches," cast-out queues for holding modified data that has been pushed out of the caches (by the replacement algorithm, cache control instructions, and/or snoop requests), and others. Hardware keeps all of these queues coherent, and in general neither software nor system hardware should be able to observe their presence.

3.2.1 Store Gathering

The 970MP microprocessor performs gathering of cacheable stores in order to reduce the store traffic into the L2 cache. The gathering occurs in L2 store queues that sit above the L2 cache. The store queue consists of eight, 64-byte wide, fully-associative entries. Stores can be gathered while architecturally permitted (that is, there is no intervening barrier operation) and the matching address is valid in the store queue. The conditions for pushing the store queue data into the L2 cache are not visible to the programmer (see *Store Processing in the CIU and L2* on page 195 for additional information).

Gathering of cache-inhibited stores is also supported and can be disabled with a mode bit in the Non-Cacheable Unit (NCU) Configuration Register.

3.3 Storage Model

3.3.1 Atomicity

The 970MP processing unit is fully compliant with the architectural requirement for single-copy atomicity on naturally aligned storage accesses.

3.3.2 Storage Access Ordering

The architecture defines a weakly ordered storage model for most types of storage access scenarios. For these cases, the 970MP microprocessor takes advantage of this relaxed requirement to achieve better performance through out-of-order instruction execution and out-of-order bus transactions. As a result, if strongly ordered storage accesses are required, software must use the appropriate synchronizing instruction (Synchronize [**sync**], Page Table Entry Synchronize [**ptesyn**], Enforce In-Order Execution of I/O [**eieio**], or Lightweight Synchronize [**lwsync**]) to enforce order explicitly, or perform these accesses to regions marked with attributes that require the hardware to enforce strong ordering (that is, stores to storage marked cache-inhibited and guarded must occur in-order).

The 970MP processing unit performs load operations out-of-order internally to the processor; however, it also keeps track of these loads in a way that lets it know when an external request for exclusivity may lead to the appearance of non-sequential execution. For these cases, the 970MP processing unit can flush potentially bad results, and re-execute the code starting from the suspect load instruction.

3.3.2.1 Storage Access Alignment Support

Most storage accesses are performed without software intervention (that is, without an alignment interrupt). The relative performance of these accesses depends to some degree on their alignment. In many cases, unaligned storage accesses are handled with a performance equivalent to aligned accesses. However, in some cases the 970MP processing unit is forced to break unaligned accesses into multiple internal operations. Further, because effective-address alignment for storage references cannot be determined until execution time, and dataflow-oriented execution pipelines of the 970MP microprocessor do not support iteration, some unaligned storage accesses actually cause a pipeline flush to allow a microcoded emulation of the instruction. For more information about the performance aspects of unaligned storage accesses, see *Section 6.3.3.3 Run Time Feedback-Based Instruction Cracking* on page 178.

The following list summarizes the cases in which the 970MP processing unit will engage a microcoded emulation of unaligned storage references:

- Any fixed-point load operation that crosses a 64-byte boundary (note 1)
- Any fixed-point load operation that misses in the L1 D-cache and crosses a 32-byte boundary (note 1)
- Any fixed-point store operation that crosses a 4-KB boundary (note 2)
- Any floating-point load double operation that is word aligned and crosses a 64-byte boundary (note 1)
- Any floating-point load double operation that is word aligned, misses in the L1 D-cache, and crosses a 32-byte boundary (note 1)
- Any floating-point store operation that is *word aligned* and crosses a 4-KB boundary (see note 1)

Notes:

1. If the instruction is not a multiple or string instruction, the access crosses a page boundary, and the access to either page causes an exception, appearing as though the load instruction has not been executed (that is, neither the **fr**D[1] or **r**D[2] is modified).
2. If the access to the first page causes an exception, storage is not modified. Otherwise, storage in the first page is updated even if the access to the second page causes an exception.

As an aid for software identification of these cases, the 970MP microprocessor supports a debug-only mode, controlled by bit 24 in Hardware Implementation Dependent Register 4 (HID4[24]), that will force an alignment interrupt in these scenarios. See *Section 4.5.8 Alignment Exception* on page 142 for a summary of cases in which the 970MP processing unit will take an alignment interrupt.

3.3.3 Atomic Updates and Reservations

The coherency granule size in the 970MP processing unit is 128 bytes. The following events will affect the state of the Reservation Register:

- Execution of a Load Word and Reserve Indexed (**lwarx**) or Load Doubleword and Reserve Indexed (**ldarx**) instruction (sets new reservation)
- Execution of a Store Word Conditional Indexed (**stwcx**) or Store Doubleword Conditional Indexed (**stdcx**) instruction (successful or not, address match or not, the reservation is cleared)
- Snooped Read with Intent to Modify (RWITM) bus operation that matches the reservation address (clears the reservation)
- Snooped Data Line Claim (DCLAIM) bus operation that matches the reservation address (clears the reservation)
- Snooped Write with FLUSH bus operation that matches the reservation address (clears the reservation)
- Snooped Write with KILL bus operation that matches the reservation address (clears the reservation)

When performing bus snooping, the 970MP processing unit checks the state of the internal caches *and* the state of the Reservation Register to formulate a snoop response. If a particular coherency block is not in any of the caches but the address is valid in the Reservation Register, then the processor and STS act as though the coherency block is in the shared state for the snoop response (this prevents another processor from taking the block as exclusive on a simple READ bus transaction).

1. Field used to specify a Floating-Point Register (FPR) to be used as a target.
2. Field used to specify a General Purpose Register (GPR) to be used as a target.

3.4 Cache Management

3.4.1 Flushing the L1 I-Cache

To help flush the entire contents of the I-cache efficiently, the 970MP microprocessor implements a special mode of operation for the Instruction Cache Block Invalidate (**icbi**) instruction. This mode can be selected using a bit in the HID1 register. In this mode, all directory lookups on behalf of an **icbi** act as though there was a real address match. Therefore, all lines looked at by the **icbi** will in fact be invalidated. As a result, the entire L1 I-cache can be invalidated by issuing a series of **icbi** instruction that step through each congruence class of the I-cache

Note: Another way to clear the I-cache is to actually fill it with a set of known values by executing a piece of code that effectively touches each line of the cache. One way to write this code is to have a series of 512 branches to branches whose effective addresses are sequentially separated by 128 bytes (the line size of the I-cache). Many other possible code sequences can achieve the same effect.

3.4.2 Flushing the L1 D-Cache

The L1 D-cache is a store-through design, so it never holds modified data. As a result, to perform a flush of the L1 D-cache, the only instruction required is a **sync**. The **sync** instruction forces any pending stores in the store queue above the L1 cache to become globally coherent before the **sync** is allowed to complete.

To completely invalidate the L1 D-cache, use the l1dc_flsh mode bit located in the HID4 to cause a flash invalidate of the D-cache. Software needs to set this bit and follow it with a **sync** instruction.

3.4.3 L2 Cache Disabling and Enabling

The L2 cache cannot be disabled.

3.4.4 L2 Cache Flushing

3.4.4.1 L2 Cache Flush in Direct-Mapped Mode

The BIU Mode Register (at SCOM address x'043000') is set to x'0000 0000 0000 8000' to enter direct-mapped mode. In direct-mapped mode, victims are selected based on a simple address decode. *Table 3-2* shows the decode. The three tag address bits used for the mapping are real address bits 42 - 44.

Table 3-2. Simple Address Decode

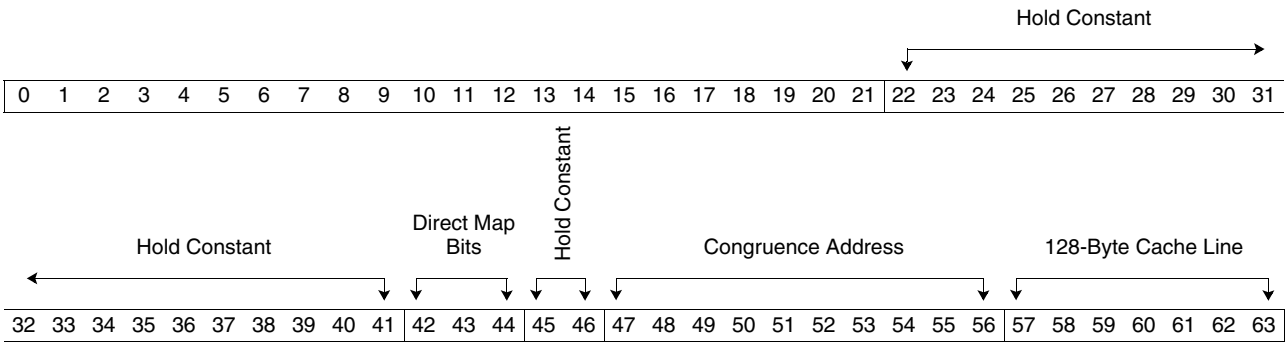
Real Address (Bits 42:44)	Selected Victim
000	A
001	B
010	C
011	D
100	E

Table 3-2. Simple Address Decode

Real Address (Bits 42:44)	Selected Victim
101	F
110	G
111	H

3.4.5 L2 Cache Flush Algorithm

L2 Address Map



Before the L2 cache of one processing unit is flushed, the other processing unit must be quiesced. For example, before flushing the L2 cache of processing unit 1, quiesce processing unit 0. Use an ATTN instruction or the service processor to quiesce the processing unit that is not being flushed (this step is not required if only one of the processing units is functional). Then, load the cache flush routine into the processing unit that is being flushed.

The following sequence will flush the entire L2 cache to memory via software:

1. Disable interrupts.
2. Disable data address translation by setting MSR[DR] to '0'.
3. Disable instruction cache (I-cache) prefetch by setting HID1[7:8] to '00'.
4. Disable data cache (D-cache) prefetch by setting HID4[25] to '1'.
5. Flash invalidate the D-cache by setting HID4[28] to '1'.
6. Execute a **sync** instruction.
7. Disable the D-cache (set HID4[37:38] to '11'). This will guarantee that all loads are visible to the L2.
8. Set the L2 to direct-mapped mode. This can be done by the service processor or through the SCOM control (SCOMC) and SCOM data (SCOMD) special purpose register (SPR) interface.
9. Execute a **sync** instruction.
10. Initialize a register with the starting address of a 4-MB cacheable region of memory that is aligned on a 4-MB boundary (that is, bits 42 - 63 are all zeros).
11. Execute eight load instructions, incrementing the direct map field (bits 42 - 44) of the load address between each load (see the *L2 Address Map*).
12. Increment the congruence address field (bits 47 - 56) of the load address, and repeat step 11 (see the *L2 Address Map*).

13. Repeat step 12 for all 1024 congruence address values.

To power down after performing this sequence, the processor executes an ATTN instruction to enter the quiescent state. Once a processing unit has been flushed, it should be fenced if the intent is to power down that processing unit. This helps avoid snooping and hang problems.

To return to normal processing after performing this sequence, set the L2 cache to set-associative mode. Enable the D-cache, prefetching, data address translation, and interrupts, as desired.

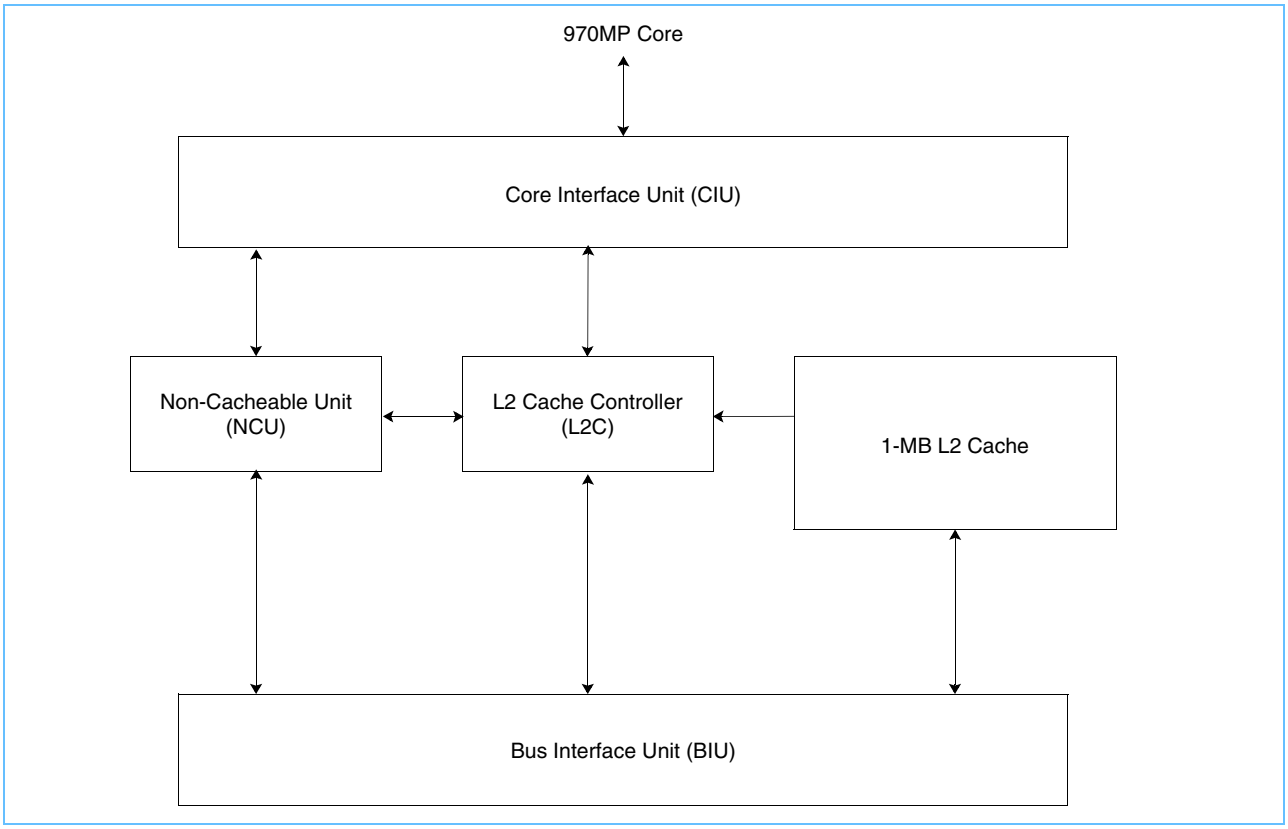
3.5 Functional Units

Table 3-3 lists the functional units within the storage subsystem, and *Figure 3-1* shows how they interact.

Table 3-3. Storage Subsystem Functional Units

Unit	Mnemonic
Core Interface Unit	CIU
L2 Cache Controller	L2C
Non-Cacheable Unit	NCU
Bus Interface Unit	BIU

Figure 3-1. 970MP Storage Subsystem



3.5.1 Core Interface Unit

The core interface unit (CIU) is the interface block between the 970MP core and the rest of the storage subsystem. It contains the necessary pipeline buffers and queues to maintain the required transfer rates to and from the 970MP core. The interface block consists of interfaces to the 970MP core, L2 cache interfaces, NCU interfaces, and reload interfaces.

The interfaces to the 970MP core include one instruction fetch unit (IFU) port, two load/store unit (LSU) ports, and one data prefetch and translate port. The CIU performs request arbitration, queueing, and flow control. It also maintains load/store ordering and provides prefetch support. In addition, the 970MP core interfaces include one store interface with the LSU. The CIU performs request queueing and flow control for this interface. It maintains store ordering and supports a 16-byte data path.

The CIU provides request flow control for the L2 cache interface. It dispatches operations to the L2 cache interface based on storage mode and operation type. The CIU also provides request flow control for the NCU interface. It dispatches operations to the NCU based on storage mode and operation type. It maintains cache-inhibited store ordering.

The reload/invalidate address interfaces include one IFU port, one LSU port, and one translate port. The CIU provides support for L1 cache invalidates. It also requests arbitration and flow control. The reload data bus is a 32-byte data path running at the CPU speed (1:1).

3.5.2 L2 Cache Controller

As shown in *Figure 3-1* on page 92, the L2 cache controller (L2C) resides between the CIU and the BIU and also interfaces with the NCU. See *Table 3-1 Storage Hierarchy Characteristics* on page 85 for additional details of the L2 cache features.

L2 Cache Features

- 1-MB size, 8-way set associative
- Fully inclusive of the L1 data cache
- Unified L2 cache controller (combines entities such as instructions, data, and PTEs)
- Store-in L2 cache (store-through L1 cache)
- Fully integrated cache, tags, and controller
- Five-state modified/exclusive/recent/shared/invalid (MERSI) coherency protocol

L2 Cache Controller Features

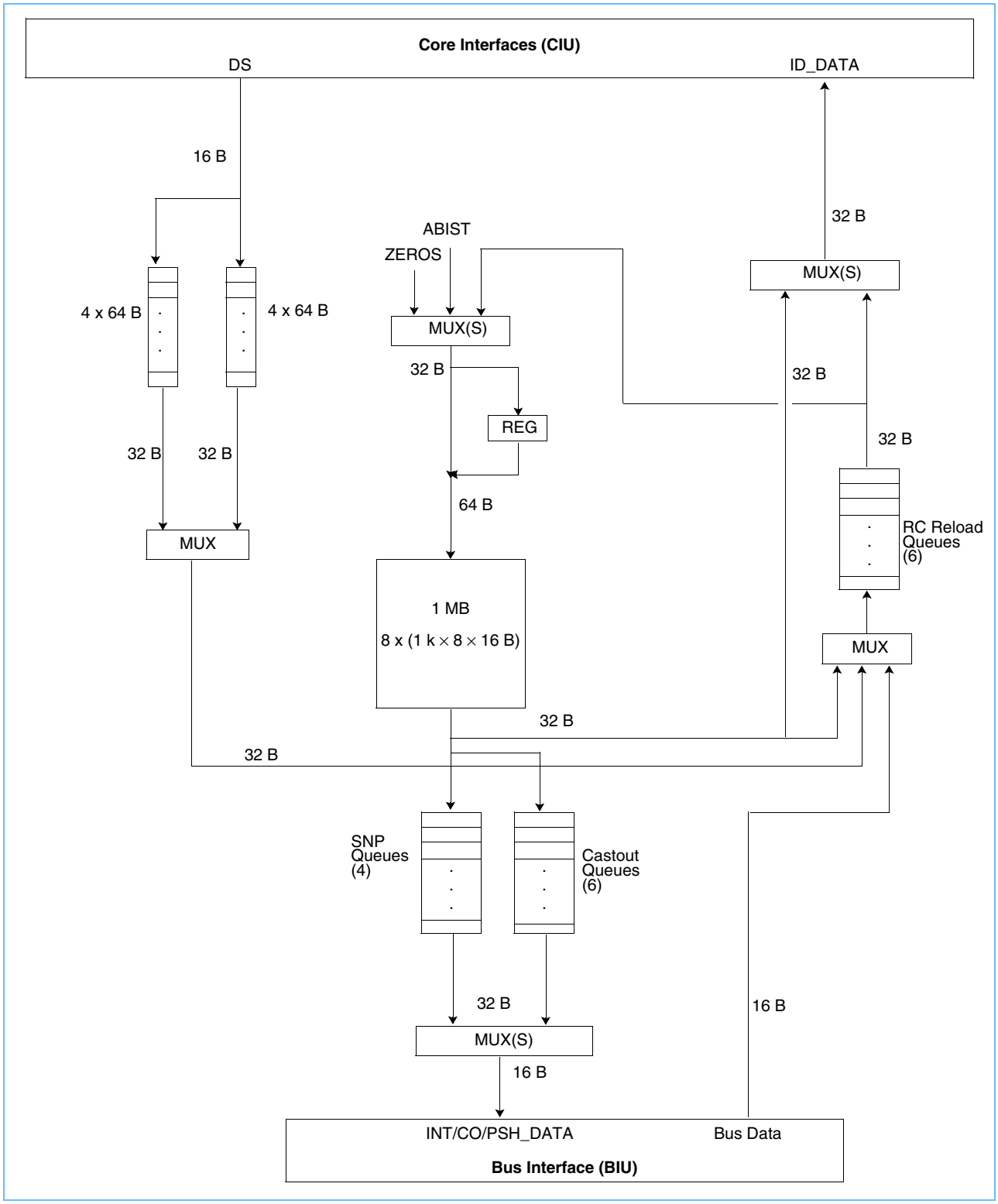
- Runs at core frequency (1:1)
- Handles all cacheable loads/stores (including **lwarx**/**stcwx**)
- Critical 32-byte forwarding on data loads
- Critical 32-byte forwarding in instruction fetches
- Six read/claim queues (RCQs)
- Eight 64-byte wide store queues
- Store gathering supported (see *Store Processing in the CIU and L2* on page 195)
- Non-blocking L1 D-cache invalidates
- Recoverable single-bit directory errors (through redundant directory)

L2 Cache Snooper Features

- Separate directory for all system bus snoops
- Four snoop/intervention/push queues

Figure 3-2 on page 94 shows the data flow of the L2 cache controller, including the data queues.

Figure 3-2. Data Flow in the 1-MB L2 Cache



3.5.2.1 Cache Coherency

The cache-coherency protocol used in the L2 cache is standard MERSI as defined in *Table 3-4*.

Table 3-4. Cache-Coherency Protocol

Status Bit	Name	Meaning
M	Modified	The cache block is modified with respect to the rest of the memory subsystem.
E	Exclusive	The cache block is not cached in any other cache.
R[1]	Recent	The cache block is shared and this processor is the most recent reader of the cache block.
S	Shared	The cache block was (and still may be) cached by multiple processors.
I	Invalid	The cache block is invalid.

1. **Implementation Note:** The 970MP microprocessor supports a cache-coherency mode in which the R state is not used. R is replaced with shared-last (SL).

3.5.2.2 Cache-Coherency Paradoxes

In the 970MP processing unit, some parts of cache-inhibited operations are handled by a special section of logic that does not access the caches as part of its normal operation. As a result, if data associated with cache-inhibited operations is present in the caches (causing a cache-coherency paradox), the 970MP processing unit will bypass some of the caches. This introduces the possibility of observing stale data (more specifically, the 970MP processing unit will read from and write to the L1 D-cache if it hits, but it will bypass the L2 cache completely).

3.5.2.3 Cache State Transition Tables

Table 3-5 and *Table 3-6* on page 96 show the cache state transitions that occur as a result of processor instructions and snooped bus operations.

Table 3-5. 970MP L2 Cache State Transitions Due to Processor Instructions (Page 1 of 2)

Number	Instruction	Storage Mode	Coherency State	Bus Operation	AResp In	Comment
1	**ld, dcb**	Ca	M, E, S, R			
2	**larx**	Ca	M, E, S, R			
3	**ld, dcbt, larx**	Ca	I → S*, E	Read Cache Line	S, Null	Atomic if LARX
4	**ld, larx,**	NonCa		Read Noncache Line		Atomic if LARX
5	**dcbt**	NonCa				No-op
6	**st, dcbtst, stcx**	Ca	M → M			
7	**st, stcx**	Ca	E → M			
8	**dcbtst**	Ca	E → E			
9	**st, dcbtst, stcx**	Ca	S, R → M	DClaim		Atomic if STCX
10	**st, dcbtst, stcx**	Ca	I → M	RWITM	RTY	Atomic if STCX
11	**st, stcx**	NonCa		Write with Flush		Atomic if STCX
12	Deallocate	Ca	M → I	Write with Kill		Copyback, W = '0', M = '0'

Note: Ca: cacheable; I = '0'. NonCa: noncacheable; I = '1'. S* means R if enabled.

Table 3-5. 970MP L2 Cache State Transitions Due to Processor Instructions (Page 2 of 2)

Number	Instruction	Storage Mode	Coherency State	Bus Operation	AResp In	Comment
13	Deallocate	Ca	E, S, I → I			
14	**dcbf**	Ca	M → I	Write with Kill		W = '1', M = '0'
15	**dcbf**	Ca	E → I			
16	**dcbf**	Ca	I, S, R → I	Flush Block		
17	**dcbst**	Ca	M→S, E	Write with Clean		Cache > E
18	**dcbst**	Ca	E, R, S→ E, R, S			→ E, R
19	**dcbst**	Ca	I	Clean		
20	**dcbz**	Ca	E, M → M			
21	**dcbz**	Ca	I, S → M	DClaim		
22	**dcbz-32byte**	Ca				32 bytes, treated as store
23	**icbi**			IKill		

Note: Ca: cacheable; I = '0'. NonCa: noncacheable; I = '1'. S* means R if enabled.

Table 3-6. 970MP L2 Cache State Transitions Due to Bus Operations (Page 1 of 2)

Number	Bus Operation		Snooper State	Rsrv State	AResp Out	AResp In	Comments
1	Read Burst	N = '1', S = '0'	M → S		M	M	Causes C → M → C data-only operation (Intervention).
2		N = '1', S = '1'	M → E		M	M	
3		N = '1'. S = '0'	E, R → S		Shrl	Shrl	Causes C → C intervention.
4		N = '1', S = '1'	E, R → E, R		Shrl	Shrl	Causes C → C intervention.
5	Read Non Burst	N = '0', S = '0'	M → S		Retry	Retry	Causes Write with Clean (Push).
6		N = '0', S = '1'	M → E		Retry	Retry	Causes Write with Clean (Push).
7		N = '0', S = '0'	E, R → S		S	S	Reader may go to R state.
8		N = '0', S = '1'	E, R → E, R		S	S	Reader will go to S state.
9	Any Read		S		S	S	
10			I	R = '0'			
				R = '1'	S		
11	RWITM	N = '1'	M → I		M	M	Causes C → C intervention.
		N = '1'	E, R → I		Shdl	Shdl	Causes C → C intervention.
12		N = '0'	M → I		Retry	Retry	Causes Write with Kill (Push).
13		N = '0'	E, R → I		Null		N says do not intervene.
14			IS → I				
15	Write-with-Kill, DKill, DClaim		IESM → I				
16	Write-with-Flush		M → I		Retry	Retry	Causes Write with Kill (Push).
17			ISE → I				

Table 3-6. 970MP L2 Cache State Transitions Due to Bus Operations (Page 2 of 2)

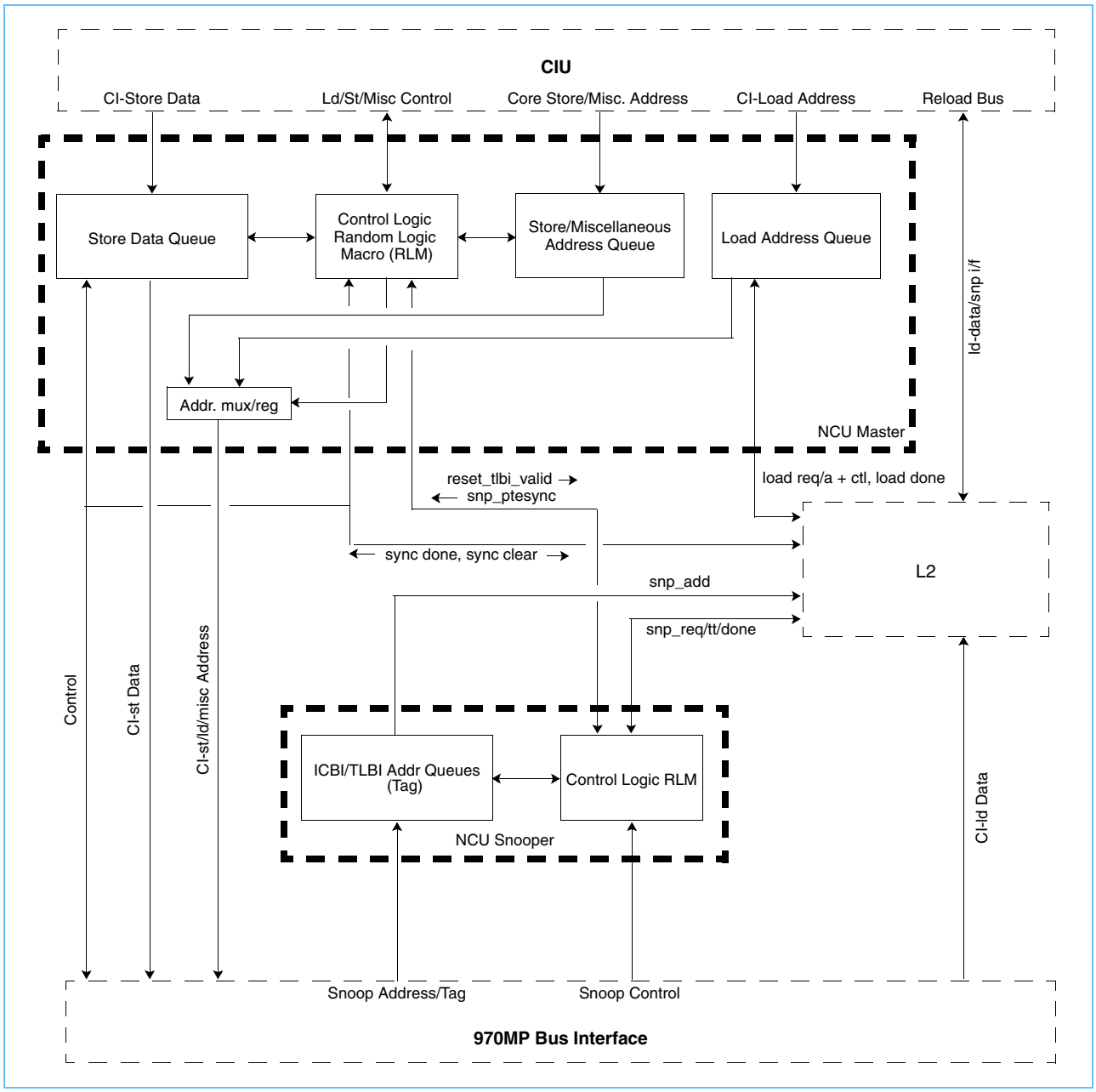
Number	Bus Operation		Snooper State	Rsrv State	AResp Out	AResp In	Comments
18	Flush		M → I		M		Causes Write with Kill (Push).
19			ISER → I				
20	Clean		M → S, E		M		Causes Write with Clean. M = '0' Cache → E
21	Clean		E, R, S → E, R, S		S		→ E, R
22	Clean		I → I		Null		
23	SYNC, TLBSYNC				Retry Null		Retry until done. Null when done.

3.5.3 Non-Cacheable Unit

The non-cacheable unit (NCU), illustrated in *Figure 3-3*, handles all communications to and from the core that are not handled by the L2 cache.

Note: The 970MP microprocessor does not support the cache-inhibited **lwarx**/**stwcx.** instruction.

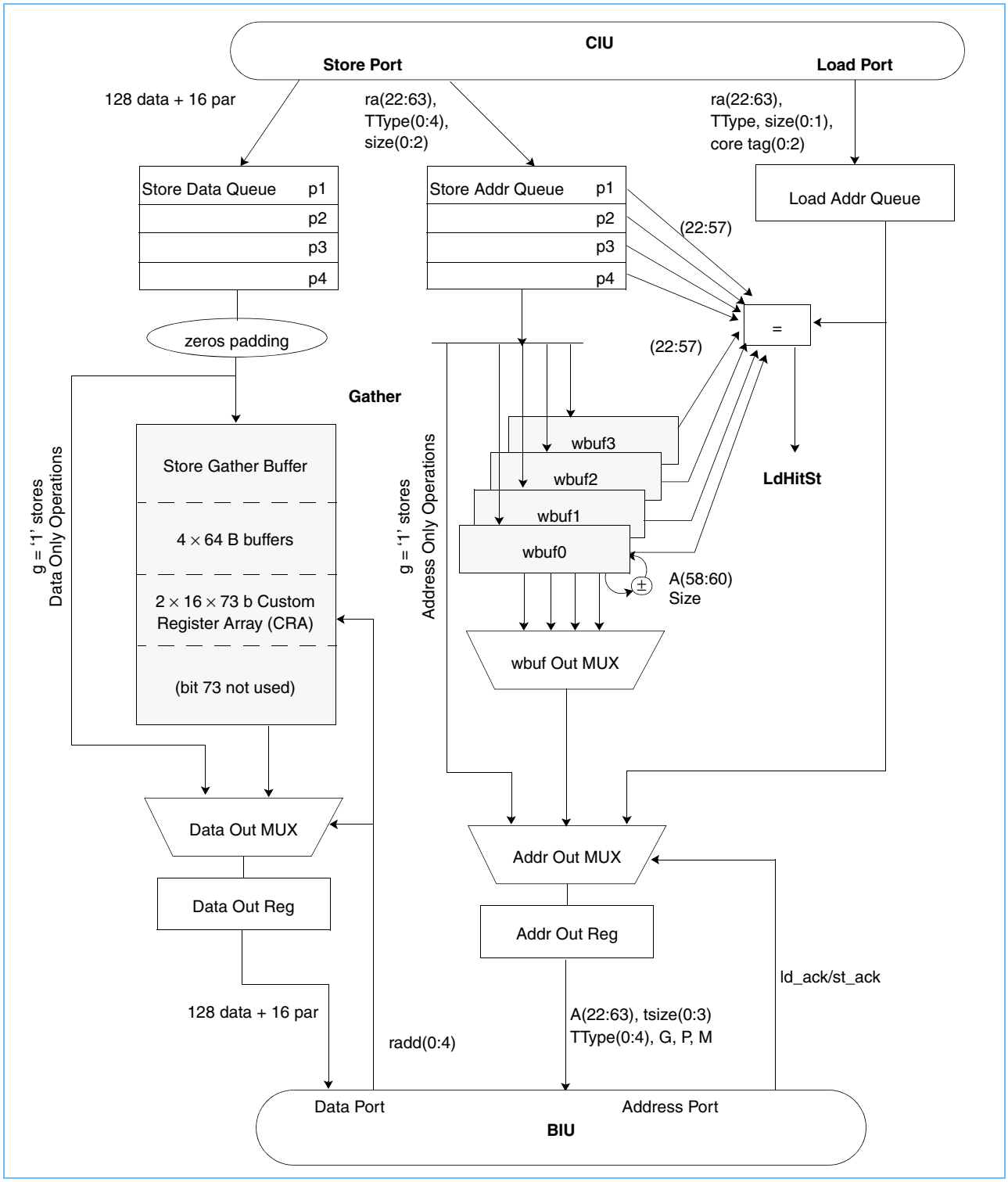
Figure 3-3. NCU Block Diagram



3.5.3.1 NCU Queues

Figure 3-4 illustrates the various queues implemented by the NCU.

Figure 3-4. NCU Master Store and Load Queues



Non-Cacheable Store Queue

The NCU implements an effective 8-entry store queue for nonguarded (G equals '0'), cache-inhibited (CI) stores. The queuing structure consists of a 4-entry, 16-byte input FIFO (labeled p1 - p4 in *Figure 3-4* on page 99) and four 64-byte gather buffers (labeled wbuf0 - 3 in *Figure 3-4*). Requests flow through the FIFO queue and are enqueued into one of the four buffers waiting to be issued to the bus. If consecutive requests can be gathered, they will be collected into the working buffer before requesting a bus transfer. A buffer will not be deallocated until all of its data is emptied to the bus without being retried.

When guarded (G equals '1'), cache-inhibited stores and address-only operations flow directly from the input queue to the bus bypassing the buffer. These operations require strict ordering. A request has to be completed on the bus without being retried before the next one can be issued. This NCU effectively provides a 4-entry queue for these types of slower operations. This type of operation remains in the queue until all gather buffers are emptied before the operation is allowed to access the system bus.

The gather buffer design is optimized to match bus performance. It can gather requests from the CIU at a peak rate of one request per processor clock cycle. A gatherable request can be of any size (1, 2, 4, 8, or 16 bytes long) as long as they fall into the consecutive byte locations.

Non-Cacheable Load Queue

The NCU implements a 1-deep load queue. Because the processor core treats all cache-inhibited loads as guarded (G equals '1') loads, it will not issue more than one load request at a time.

The load queue checks against all the current stores in the store queue for an address hit. A snapshot of all the valid store queue entries is taken for address comparison at the time the load request is received. If a hit is detected, the load is rejected (with no acknowledgment to the CIU) and retried by the CIU until accepted.

Snoop Queue

The NCU snooper implements two sets of snoop queues, one for Instruction Cache Block Invalidate (**icbi**) instructions and one for Translation Lookaside Buffer Invalidate (**tlbi**) instructions. Each queue is two entries deep in order to support two back-to-back requests of each type.

A snooped SYNC operation checks against the processor identification tag (PID) associated with the ICBI and TLBI requests queued. If a hit is detected (that is, the SYNC is issued by a processor that has a pending ICBI or TLBI requests), the SYNC is retried.

3.5.3.2 Downstream Operations

Non-Cacheable Store

Non-cacheable stores issued from the core to the NCU can have a granularity of 1, 2, 4, 8, and 16 bytes. Furthermore, multiple stores can be gathered up to 64 bytes. The rules for gathering are:

- Gathering starts at a 64-byte boundary.
- Only the next consecutive bytes will be gathered.
- Gathering stops at a 64-byte boundary.
- Guarded (G equals '1') store and address-only operations will break the gathering loop.
- Gathering window controls are modeable. A mode bit (ncs_cfg_gth_tmr_off) can disable the gather timer. When disabled, the gathering will not time-out unless a load-hit-store collision is detected. The gather

timer is 5 bits, with the upper 3 bits programmable (ncs_scom_cfg_gth_tmr_limit). The default of '11000' yields a count of 24. The counter resets on every write to the NCU write buffer. The gathering window is extended past the timeout if there are three (out of four) NCU gather buffers waiting for bus operations, or if there is a gatherable store pending in any of the four NCU store input pipe stages.

- Gather must be enabled by the configuration mode bit.

Note: The store gathering logic only gathers bytes that fall into the next consecutive byte of the address computed.

Non-Cacheable Load

Non-cacheable loads issued from the core to the NCU can have a granularity of 1, 2, 4, 8, and 16 bytes for data fetch. Instruction fetch is limited to 32-byte loads only. The return of cache-inhibited load data and instructions to the core is not routed through the NCU, but is returned through the L2 cache by the same reload bus used for cacheable loads.

Note: Cache-inhibited loads are treated as guarded (G equals '1') loads by the core. They are issued sequentially; that is, a cache-inhibited load must be completed with data returned before the next cache-inhibited load can be issued.

PowerPC Cache and Barrier Operations

The PowerPC cache and synchronization operation instructions are sent downstream to the NCU. The NCU enters the operations into its non-cacheable store queue just like stores. The difference is that there is no data associated with these operations. They are issued out onto the bus. Most of these operations will be snooped back up by the L2. These operations include:

- Instruction Cache Block Invalidate (ICBI): Snooped by the NCU snooper, but not by the L2.
- TLB Invalidate (TLBI): Snooped by the NCU snooper, but not by the L2.
- TLB Synchronization (TLBSYNC): Treated as a no-op by the NCU; not issued to the bus.
- Instruction Cache Synchronize (ISYNC): This operation is never seen outside the 970MP core.
- Enforce In-Order Execution of I/O (EIEIO): Takes the store queue slot in the L2 *and* in the NCU.
- Synchronize (SYNC): Takes the store queue slot in the L2 *and* in the NCU.
- Page Table Entry Synchronize (PTESYNC): Takes the store queue slot in the L2 *and* in the NCU.
- Lightweight Synchronize (LWSYNC): Takes the store queue slot in the L2 *and* in the NCU.
- Data Cache Block Store (DCBST): Takes a store queue slot in the L2 only; not sent to the NCU.
- Data Cache Block Flush (DCBF): Takes a store queue slot in the L2 only; not sent to the NCU.
- Snoop TLB Complete (SnpTLBIcmp): The core acknowledges that a snooped TLBI has completed.

PTESYNC, SYNC, LWSYNC, and EIEIO all create a barrier in the NCU and the L2 cache, but the NCU snooper treats them differently. The following rules are observed by the NCU snooper:

- PTESYNC: Respond with a snoop retry if a pending ICBI or TLBI request from the same processor is detected.
- SYNC: Respond with a snoop retry if a pending ICBI request from the same processor is detected.
- EIEIO: The snooper ignores this bus operation.

- LWSYNC: The NCU master does not issue this operation to the bus by default. The NCU snooper always ignores this bus operation.

ICBI and TLBI are snooped by all NCU snoopers in a multi-processor system, so that they can be propagated to the cores. These snoops are sent to the core through the L2 reload bus interface.

3.5.3.3 Upstream Operations (Snoops)

The NCU (snooper) handles the snooping of the following operations:

- ICBI
- TLBI
- PTESYNC and SYNC

A snooped TLBI that was issued by the local core (that is, the core supported by this NCU) is treated as a no-op. The barrier operations (PTESYNC and SYNC) are snooped only because they may have to be retried due to any incomplete ICBI or TLBI that was issued with the same processor ID as the barrier. When a PTESYNC is snooped and it completes the snoop cycle without being retried, a PTESYNC pulse is generated to the core.

3.5.3.4 Transfer Type Mapping

Table 3-7 maps the transfer type (TType) received by the NCU on the store port to the TType issued by the NCU. *Table 3-8* maps the TType received by the NCU on the load port to the TType issued by the NCU.

Table 3-7. CIU to Processor Interconnect Bus Transfer Type Mapping (Store Port)

CIU TType Received by NCU (Store Port)	Processor Interconnect Bus TType Issued by NCU				
Operations	TType(0:4)	M[1]	P[2]	G[3]	Operations
CI STORE TSIZE(0:2) 000 = 1 B 001 = 2 B 010 = 4 B 011 = 8 B 1xx = 16 B	00010	0	1	g	Write with Flush TSIZ(0:3) 0001 = 1 B 0010 = 2 B 0100 = 4 B 1000 = 8 B 1010 = 16 B 1100 = 32 B 1101 = 64 B
DCBF	00100	1	1	g	Flush
ICBI	01101	1	0	g	IKill
PTESYNC	01000	0	1	g	SYNC with TSIZ(0) = '1'
EIEIO	10000	0	1	g	EIEIO
LWSYNC	No Bus Operation				
SYNC	01000	0	1	g	SYNC with TSIZ(0) = '0'
TLBSYNC	01001	0	1	g	TLBSYNC
TLBIE	11000	0	0	g	TLBIE

1. M = memory coherent
2. P = pipelined snoop
3. "g" indicates pass along the G bit value set by the core.

Note: In store gather cases, the NCU master will break up gathered bytes into multiple bus transactions to fit into allowable transfer sizes.

Table 3-8. CIU to Processor Interconnect Bus Transfer Type Mapping (Load Port)

CIU TType Received by NCU (Load Port)	Processor Interconnect Bus TType Issued by NCU				
Operations	TType(0:4)	M[1]	P[2]	G[3]	Operations
CI Data Fetch TSIZE(0:2) 000 = 1 B 001 = 2 B 010 = 4 B 011 = 8 B 1xx = 16 B	01010	0	1	1	Read TSIZ(0:3) 0001 = 1 B 0010 = 2 B 0100 = 4 B 1000 = 8 B 1010 = 16 B
CI Instruction Fetch Fixed-size 32-byte transfer	01010	0	1	1	Read with TSIZ(0:3) = 1101 (32 B)

1. M = memory coherent
2. P = pipelined snoop
3. "g" indicates pass along the G bit value set by the core.

Synchronization Operations

The NCU handles two classes of synchronization operations. One class, referred to as barrier operations (PTESYNC and SYNC), requires a **sync** complete to be sent back to the core. The other class (LWSYNC and EIEIO) does not require a core response, but does have ordering rules.

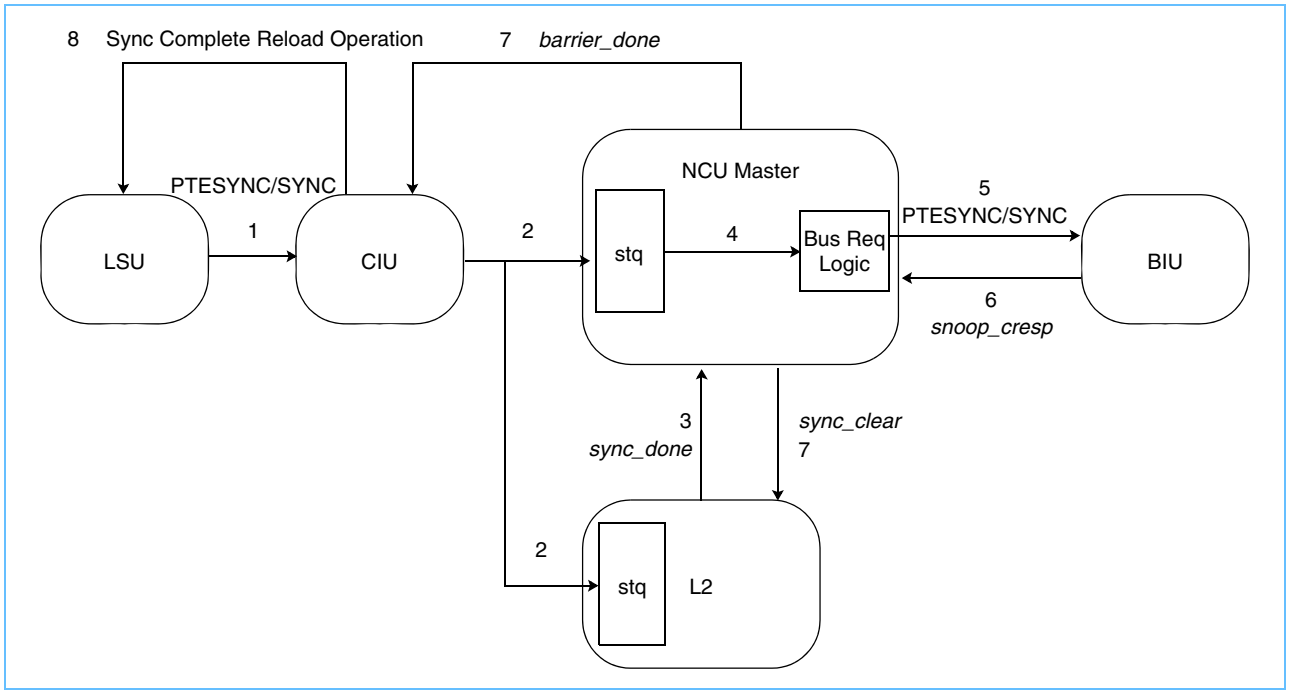
Barrier Operations

PTESYNC and SYNC have the strictest ordering requirements in L2 and NCU load/store queues. Loads and stores issued before a barrier operation, must be completed before younger loads and stores issued after the barrier operation can proceed. When a barrier operation reaches the bottom of the NCU store queue, it waits there until all previous cache-inhibited stores are completed and the L2 finishes its synchronization. The NCU then issues the synchronization command on the bus. An external controller (the memory controller in most cases) will broadcast this command to all processors. Upon receiving the snoop command, the processors will generate an appropriate snoop response based on their internal states (see *PowerPC Cache and Barrier Operations* on page 101). The NCU will not release the barrier until the bus synchronization command has completed successfully (not retried).

Note: The NCU does not check for any outstanding cache-inhibited loads that are queued. Because a cache-inhibited load is treated as a guarded (G equals '1') operation, the core must wait until the cache-inhibited load finishes before it can issue a barrier operation.

PTESYNC and SYNC

Figure 3-5. NCU Master Barrier Operations Flow



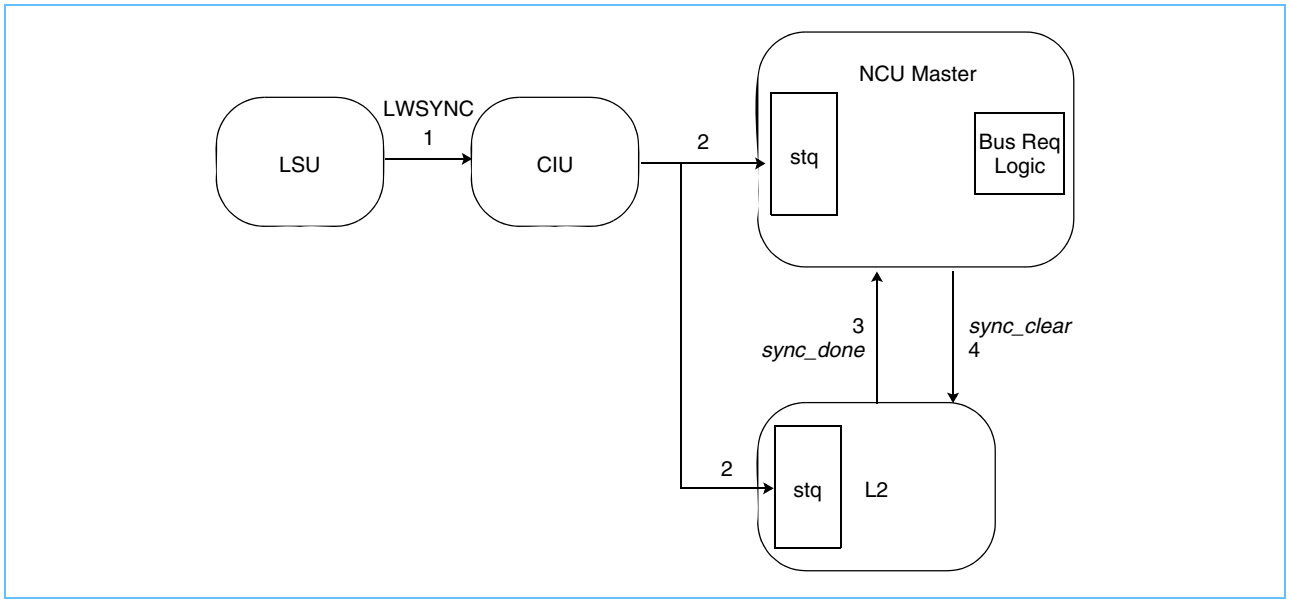
NCU Handling of PTESYNC and SYNC (see *Figure 3-5*)

1. The LSU issues a PTESYNC/SYNC barrier operation to the CIU. No additional stores will be sent be the LSU until it receives the **sync** complete. However, speculative loads may be sent by the LSU.
2. The CIU forwards the command to the NCU and L2 through their store interfaces. The NCU enqueues the barrier operation in its store queue (FIFO) and creates a barrier. The L2 also creates a barrier in the cacheable store queue.
3. When the barrier operation reaches the bottom of the NCU input store queue, it waits until all previous NCU stores are completed. It also waits for the L2 to complete its stores issued before the barrier operation. The L2 signals its completion by sending *sync_done* to NCU.
4. When both the L2 and NCU have completed all older store operations, the NCU requests the bus to broadcast the barrier operation.
5. When the NCU receives a grant from the BIU, it sends a SYNC bus command (with M equal to 1' and P equal to '1') to the BIU. TSIZ(0) is encoded to differentiate PTESYNC (equals '1') and SYNC (equals '0').
6. The NCU then waits for the snoop bus response. If the bus transaction is retried, the NCU will reissue the transaction until it completes successfully on the bus; that is, until it is not being retried.
7. When the SYNC bus transaction completes successfully, the NCU terminates the operation, and sends a *barrier_done* signal to the CIU. The NCU also send *sync_clear* back to the L2, and the barrier is removed.
8. The CIU then compiles a *sync complete* message back to the LSU through the reload bus.

LWSYNC

LWSYNC is a light-weight synchronization operation that requires ordering on cacheable loads and stores. It operates similarly to the barrier operations except that **sync** complete is not sent back to the LSU and the LSU can send additional load or store commands after sending the LWSYNC.

Figure 3-6. NCU Master LWSYNC Operations Flow



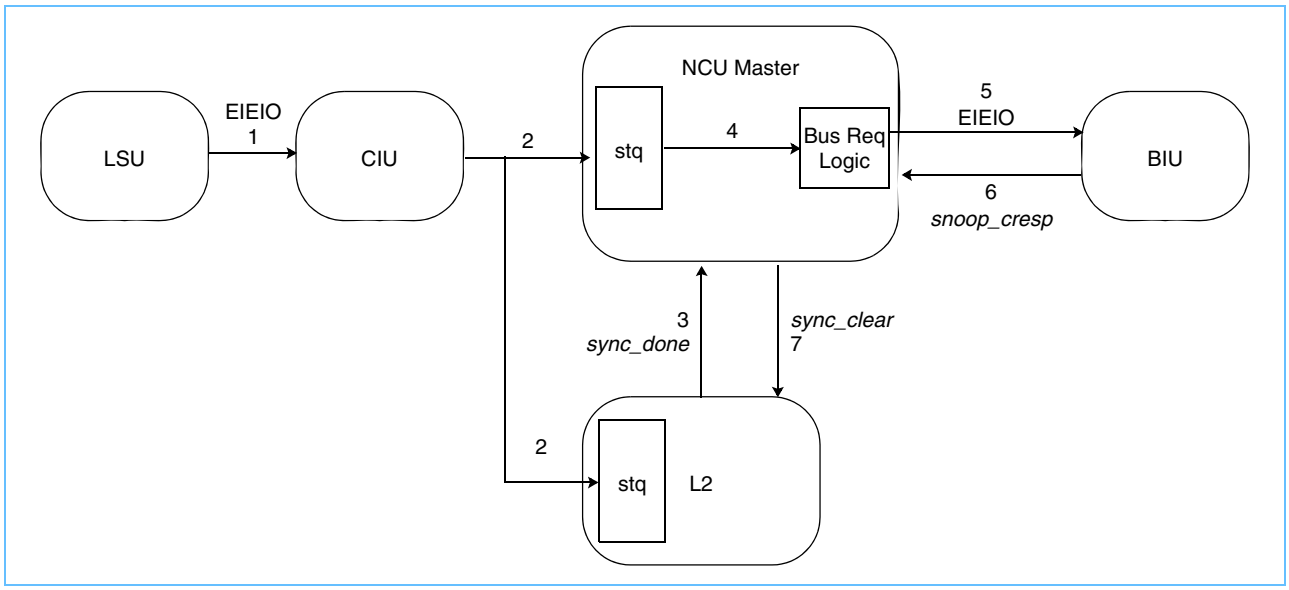
NCU Handling of LWSYNC (see *Figure 3-6*)

1. The LSU issues an LWSYNC barrier operation to the CIU.
2. The CIU forwards that operation to the NCU and L2 through their store interfaces. The NCU enqueues the barrier operation in its store queue (FIFO) and creates a barrier. The L2 also creates a barrier in the cacheable store queue.
3. When the barrier operation reaches the bottom of the NCU input store queue, it waits for the L2 to complete its older stores issued before the barrier operation. The L2 signals its completion by sending *sync_done* to the NCU. (There may still be cache-inhibited stores in the gather buffer, but these are not required to be ordered by the LWSYNC.)
4. The NCU completes the operation by sending *sync_clear* back to the L2, and the barrier is removed.

EIEIO

EIEIO operates like the LWSYNC, except that ordering is required for cacheable stores and cache-inhibited loads and stores. The LSU can send additional load or store commands after sending the EIEIO.

Figure 3-7. NCU Master EIEIO Operations Flow



NCU Handling of EIEIO (see *Figure 3-7*)

1. The LSU issues an EIEIO operation to the CIU.
2. The CIU forwards that to the NCU and L2 through their store interfaces. The NCU enqueues the barrier operation in its store queue (FIFO) and creates a barrier. The L2 also creates a barrier in the cacheable store queue.
3. When the barrier operation reaches the bottom of the NCU input store queue, it waits until all previous NCU stores are completed. It also waits for the L2 to complete its stores issued before the barrier operation. The L2 signals its completion by sending *sync_done* to the NCU.
4. When both the L2 and NCU have completed all older operations, the NCU requests the bus to issue the EIEIO.
5. When the NCU receives a grant from the BIU, it sends an EIEIO bus command (with M equal to '1' and P equal to '1') to the BIU.
6. The NCU then waits for the snoop bus response. If the bus transaction is retried as notified by the BIU, the NCU will reissue the transaction until it completes successfully on the bus; that is, until it is not being retried.
7. When the EIEIO bus transaction completes successfully, the NCU terminates the operation, sends a *sync_clear* signal to L2, and the barrier is removed.

ICBI

The 970MP processing unit offers a 2-deep ICBI request queue to address ICBI performance issues in a 2-way to 4-way symmetric multiprocessor (SMP) system.

NCU Master Handling of ICBI

1. When the LSU issues an ICBI, it increments its outstanding ICBI counter. All Instruction Cache Synchronize (ISYNC) operations are held up whenever this counter is non-zero (outside the NCU's scope).
2. The CIU forwards the operation to the NCU master through the data store interface. The NCU master queues the ICBI in its store queue (FIFO).
3. When the ICBI reaches the bottom of NCU store queue and if all previous stores are completed, the queue requests to ship the ICBI to the bus.
4. When the BIU grants the NCU's request, the NCU packs the ICBI into an Instruction Line Kill (IKILL) bus transaction (with M equal to '1' and P equal to '0') and sends it to the BIU.
5. When the ICBI has received a null response (not retried) from the bus, the operation is considered complete from the perspective of the NCU masters. If the bus operation is retried (notified by the BIU), the NCU master will reissue the transaction until it completes successfully on the bus (that is, until it is not being retried).

Note: The flow is same as shown in *Figure 3-7 NCU Master EIEIO Operations Flow* on page 106.

NCU Snooper Handling of Local ICBI

1. When a local ICBI is snooped by the local NCU snooper, it will try the operation again if the ICBI request queue is full. If the queue is not full, it will enqueue the request (cache line address and tag) if the bus snoop response is good (that is, no retry).
2. If the combined snoop response is a retry, the operation is aborted and the queue is freed up. If the combined response is a null, a request is sent to the L2, which will forward the snoop ICBI to the core (IFU) with a flag set to mark this ICBI as locally sourced. While the ICBI queue is busy, a snooped SYNC with a bus tag PID that matches that of the ICBI in progress will be retried.
3. When the reload bus operation (ICBI) is completed between the L2 and CIU, a *snp_done* pulse is sent back from the L2 to the NCU snooper. The operation is now completed with respect to the NCU snooper.
4. When the IFU completes executing the snooped ICBI, it sends a pulse to the LSU, which decrements its outstanding ICBI counter. This pulse is only sent for a locally sourced ICBI.

NCU Snooper Handling of Remote ICBI

A remote ICBI (that is, not sourced locally) is handled exactly the same as the local ICBI described above except that the ICBI request is not marked locally sourced when sent to the IFU.

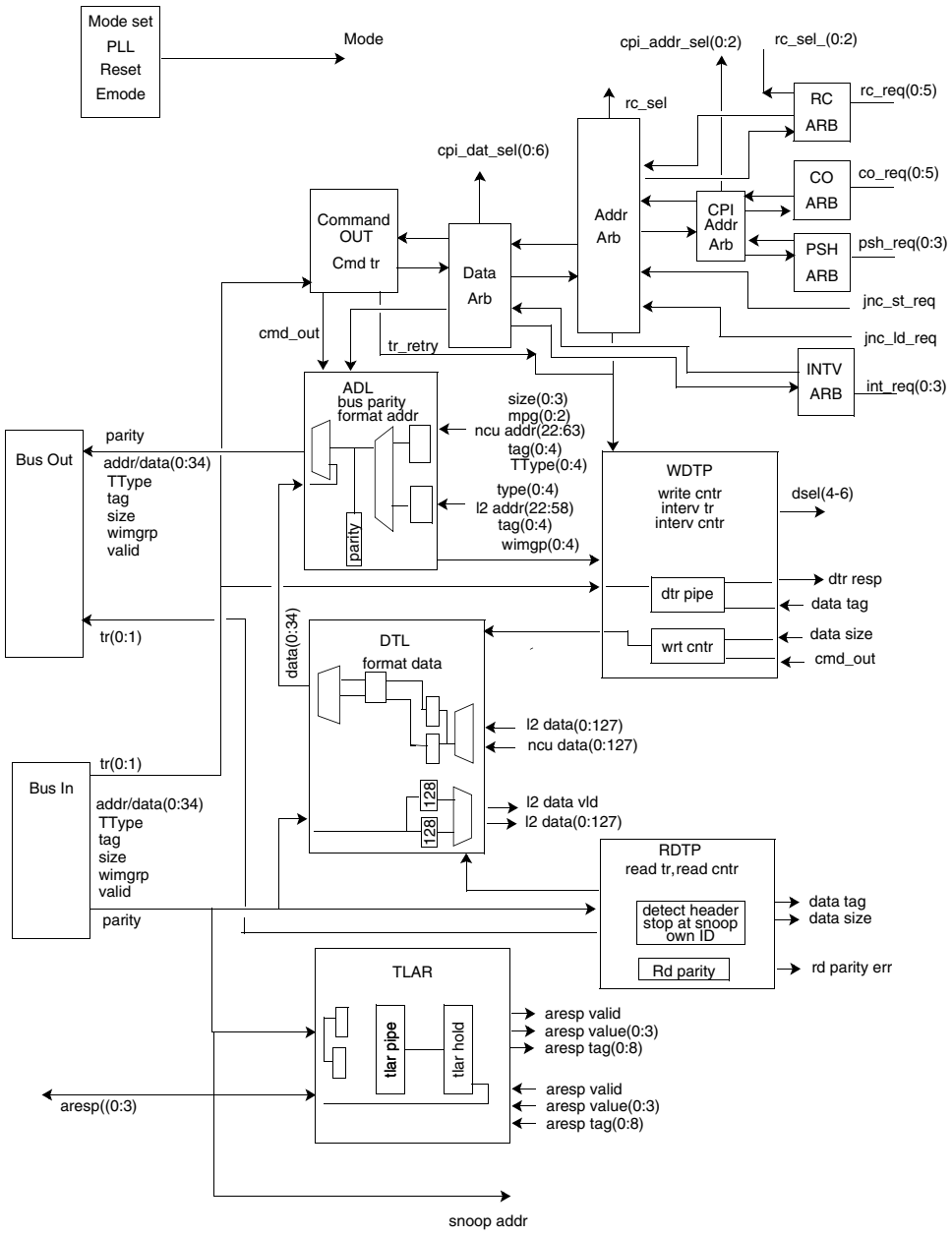
Note: The NCU snooper will pace ICBI requests sent to the IFU so that it will not exceed a rate of one every 16 N:1 clock cycles. This is a limit posted by the core. This limit is programmable (scan only latch) with a range of 0 - 8 cycles.

3.5.3.5 Bus Interface Unit

The processor interconnect bus is used to attach high-performance PowerPC processors to memory and high-speed I/O devices. It is a high-frequency, point-to-point bus. Because of its high frequency, only one attachment per segment is allowed (point-to-point connections). See the *IBM PowerPC 970MP RISC Microprocessor Datasheet* for more details.

The bus interface unit (BIU) contains the logic required to arbitrate among the various STS bus requesters and to present these requests to the bus. The BIU also contains the logic to decode transaction responses and snoop responses and forward these results back to the STS bus requester. Similarly, the BIU decodes and passes bus snoop transactions to the STS snoop logic. This level of the STS contains no queued requests other than the bus arbitration requests. All coherency-control and retry logic exists in the bus requesters. *Figure 3-8* on page 109 shows the address and data flow of the BIU.

Figure 3-8. BIU Address and Data Flow



3.5.4 Data Prefetch

The 970MP processing unit supports two versions of data prefetching. The first is a mechanism primarily controlled by hardware, which is hardware paced. The second is a subset of the vector prefetch instructions, which are software paced. Both of these mechanisms use essentially the same prefetch hardware. *Section 3.5.4.1* through *Section 3.5.4.3* describe the hardware data prefetching mechanisms and the associated support for software control of this hardware. *Section 3.5.4.4 Vector Prefetch Instruction Support* on page 123 describes the 970MP support for vector prefetch instructions.

3.5.4.1 Overview of the Hardware-Controlled Data Prefetch

The purpose of the data prefetch mechanism is to reduce the negative performance impact of increasing memory latencies, particularly on technical workloads. These programs often access memory in regular, sequential patterns. Their working sets are also so large that they often do not fit into the L1 and L2 caches used in the 970MP processing unit.

Designed into the load/store unit, the prefetch engine can recognize sequentially increasing or decreasing accesses to adjacent cache lines and then request anticipated lines from more distant levels of the cache and memory hierarchy. The usefulness of these prefetches is reinforced as repeated demand references are made along such a path or stream. The depth of prefetch is then increased until enough lines are being brought into the L1 and L2 that much or all of the load latency can be hidden, or until the processor memory bandwidth limits are reached. The most urgently needed lines are prefetched into the nearest cache level. During stream start up, several lines ahead of the current demand reference can be requested from the memory subsystem. Once steady state is achieved, each stream will bring one additional line into L1 and one additional line into L2.

Hardware Prefetch Operation (Basic Prefetch Start-Up Sequence)

Prefetch begins by saving the real address of the L1 D-cache misses in a 10-entry filter queue, offset up or down by one line address. A subsequent L1 miss (call this line *n*) that matches a filter entry establishes a stream entry in the prefetch engine, kicking off the first set of prefetches, which are initially the next line (*n + 1*) to L1 and line *n + 2* to L2. Another confirmation occurs when execution generates a demand read of line *n + 1*. This triggers the prefetch engine to request that line *n + 2* be fetched from L2 to L1, and *n + 3* be brought into L2 along with lines *n + 4* and *n + 5*. Upon still further confirmation (a demand read of *n + 2*), line *n + 3* is brought into L1, *n + 6* and *n + 7* are requested from L2. The full prefetch depth is reached when, upon a demand read of *n + 3,* line *n + 4* is brought into the L1, and line *n + 8* is brought into the L2.

The direction is based on the position of the load address within the line. (While this is somewhat arbitrary, an initial direction must be chosen, and always going up is considered less likely to provide the best performance.) Subsequent references either confirm the initial direction or establish another filter entry for the opposite direction with the next related reference establishing the stream and its actual direction. If the initial address falls in the bottom 3/4 of the line (bytes 0:95), then the direction is guessed up. If the initial address is in the upper 1/4 of the line (bytes 96:127), then the direction is guessed down.

The filter queue is updated on an LRU basis whenever an L1 miss occurs. Streams do not cross memory page boundaries, and they remain active until replaced with a new address by the filter mechanism.

In all cases, a prefetch request is completed when the lines are found already resident in or returned to the target cache level. If a demand miss catches up with an outstanding prefetch request, it is either merged or rejected depending on the level in the cache hierarchy where the collision occurs.

Data Prefetch Features

The data prefetch mechanism consists of eight streams. Prefetching occurs only on load requests; store prefetching is not performed. Prefetching and allocation takes place at both L1 and L2 cache levels. The data prefetch mechanism prefetches and allocates up to one cache line ahead to the L1 cache. It prefetches up to four cache lines ahead to the L2 cache.

L2 Prefetch Requests

Prefetch requests are flagged as special requests by the LSU. The L2 cache receives all line prefetch requests and checks its directories as it would for a demand read request. If the LSU has requested the head (first line) of the stream, the line is returned to the LSU as for any other demand read. Other prefetch-specific requests are intended to bring the lines past the head for that stream into the L2. The actions taken for the L2 prefetch requests can be summarized as follows:

- L2 Hit: No further action.
- L2 Miss: Line is requested from memory and allocated to L2 without being forwarded to the LSU.

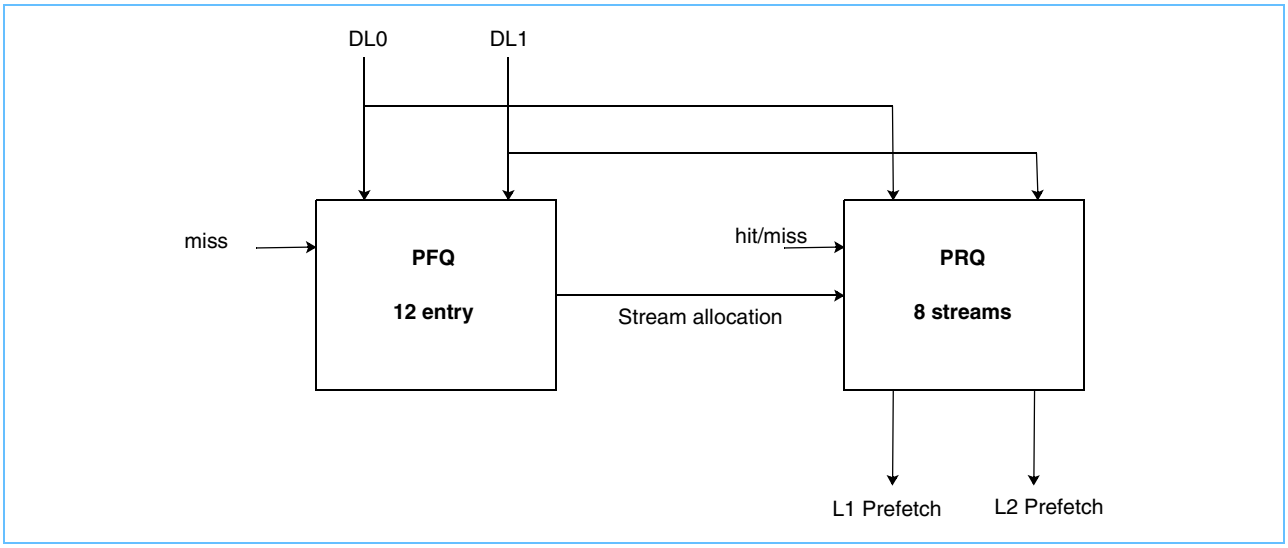
Software Touch Instruction

A special version of the existing Data Cache Block Touch (**dcbt**) instruction lets software bypass hardware detection of the stream. By setting two of the reserved bits in this instruction (one that means this is a stream touch, and one that indicates the direction), software can immediately start a stream. Once the stream is started, normal aging algorithms apply as if the instruction was initiated by hardware. A special allocation replacement algorithm guarantees an LRU replacement for streams initiated by software. See *Optional dcbt Variant* on page 120 for further information.

3.5.4.2 Hardware Prefetch Engine Implementation

The hardware prefetch engine contains logic for stream detection and the general control logic for sending commands for prefetching to each level of cache. This logic is integrated in the LSU as shown in *Figure 3-9*. There are two data demand ports from the LSU to the CIU: DL0 and DL1. The prefetch engine consists of two parts: the data prefetch filter queue (PFQ) and the data prefetch request queue (PRQ). The PFQ contains the logic to detect individual streams and then allocate those streams to the PRQ. The PRQ controls the prefetching of cache lines in anticipation of their use by the load/store unit.

Figure 3-9. Prefetch Engine Block Diagram



Data Prefetch Filter Queue

The PFQ, illustrated in *Figure 3-10* on page 115, is a 12-entry compare structure used to detect streams and prevent thrash. Real addresses are used for stream detection, which occurs after two consecutive misses. Two real addresses, one from each LSU, are presented per cycle to the PFQ. The stream direction is guessed based on the first entry into the table, and is self-correcting on short-stride streams, where the stride equals 1, 2, and so on.

Using a 2-stage pipe (compare and update, and allocate), the PFQ allocates streams and maintains stream allocation. Entries are aged out of the table based on a least recently written algorithm. This includes entries that are marked as valid streams, as well as random types of entries.

A 6-entry deep queue, (not shown), queues up to two requests per cycle, and issues up to one request per cycle to the PFQ.

The rules for PFQ management are as follows:

- A valid real address for an L1 miss is compared in content-addressable memory (CAM). When no match occurs or when a match occurs but a stream has not been allocated to that entry, the next LRU entry is loaded with:
  - (Cache line address + 1) if bits 57:58 of the address are not '11'; the stream direction is guessed up.
  - (Cache line address - 1) if bits 57:58 of the address are '11'; the stream direction is guessed down.

  If a match did not occur, and the real address of the L1 miss has a stream ID associated with it (L1 prefetch miss), then the stream tag for the next LRU location is also updated with that stream ID and that stream's direction.
- A valid real address for an L1 miss is compared in CAM. When a match occurs, the entry that matched is loaded with:
  - (Cache line address + 1) if the direction bit of the matched entry is up.
  - (Cache line address - 1) if the direction bit of the matched entry is down.

  If this entry is not marked as a valid stream, and a stream is available, then the next available stream tag is assigned. The address and tag number are scheduled for transfer to the PRQ. The entry is marked as a valid, allocated stream.

One allocation can be done per cycle.

Notes:

- If all streams are assigned, a new stream is not assigned.
- A valid L1 miss address is defined as one that is not guarded (G equals '0') and not cache inhibited (I equals '0').
- If a filter entry crosses a 4-KB or 16-MB page boundary, it will not be written into the filter.
- The next available stream tag assignment algorithm is shown below.

Algorithm for Next Filter Entry Assignment

As shown in *Figure 3-10* on page 115, a true LRU algorithm is implemented for the 12-entry queue. The filter also updates the LRU to the most recently used.

Algorithm for Stream Assignment

A simple algorithm is used for the hardware initiated stream:

```
If stream0 is not assigned then select stream 0
   else if stream1 is not assigned then select stream1
       else if stream2 is not assigned then select stream2
       and so on...
```

For the stream initiated by software, a FIFO round-robin stream is used.

Note: When a stream initiated by software is allocated, it is possible that it will overwrite the most recently allocated stream initiated by hardware.

Streams Initiated by Software

Streams initiated by software are allocated as soon as the **dcbt** instruction is presented to the filter. In these cases, the **dcbt** fetches the first cache line of the stream. The filter recognizes this transaction as a software initiated stream, allocates the stream to the PRQ, and begins prefetching at the next cache line address. The next address is also placed in the filter much like a hardware allocated entry (that is, the LRU is updated, a stream ID is assigned, and so on). Once the stream is initiated, all normal rules apply.

Note: If a **dcbt** hits the cache, no prefetching will be done.

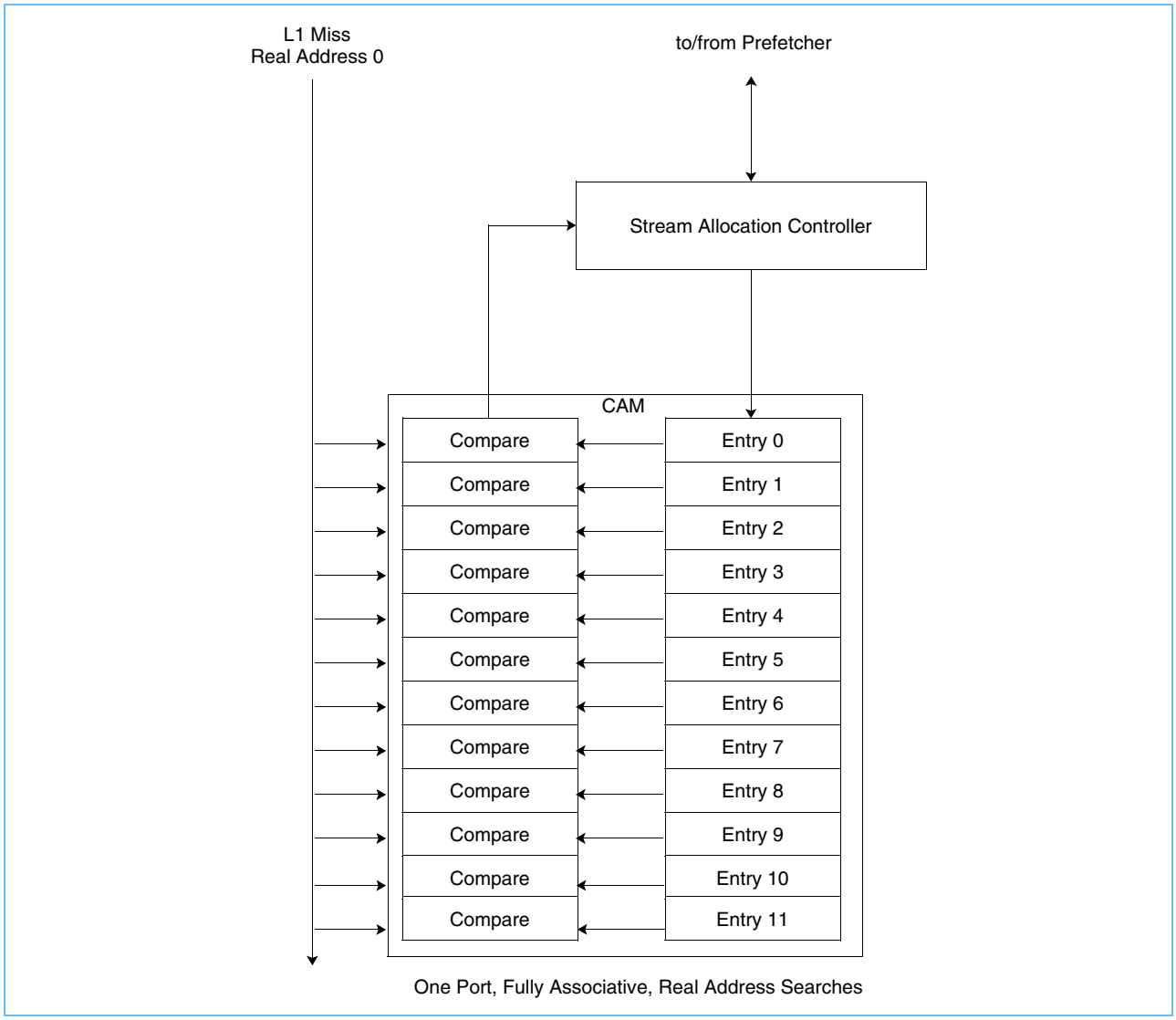
Special Cases of Stream Detection

The hardware is not capable of detecting streams where load requests occur in the same cycle (that is, in the *n* and *n + 1* address generation stages [AGENs] within the same cycle). However, hardware is able to detect these cases when they occur in adjacent cycles.

Special Cases of Stream Continuation

When an L1 prefetch occurs and its original filter entry has aged out of the filter, the stream tag ID of the L1 prefetch is placed in the filter queue along with the address at the next LRU location, and a new stream will not be initiated. This mechanism allows stream continuation when a tagged entry has been aged out of the filter queue.

Figure 3-10. Prefetch Filter Queue Logic



Data Prefetch Request Queue (PRQ)

The PRQ, illustrated in *Figure 3-11* on page 119, supports eight streams. Real addresses are used to trigger prefetches.

L1 prefetch requests enter the LSU pipeline in stage 2 (ACC) through the LSU0 execution unit. These requests, which block address generation (AGEN) by rejecting the issuing operation, have highest priority. Then, the prefetch requests enter the load miss queue (LMQ). The LMQ can reject an L1 prefetch request. If it does so, reissue will not occur (the prefetch request is dropped). Next, the L1 prefetch requests are sent to the L2 cache through the normal reload port. One cache line of prefetch instructions per stream is allocated in the L1 cache.

L2 prefetch requests enter the pipeline of the 970MP processing unit through a bus dedicated to the L2 that is shared only with translation requests. Translation requests have highest priority. Four cache lines of prefetch data per stream are allocated to the L2 cache.

Ramping up the streams is a trade-off between ramping up as soon as possible for long streams compared to not ramping up at all for short streams. Because the 970MP processing unit requires many lines of prefetch to reach full memory bandwidth, a short stream may be completed before full ramp up is achieved. Therefore, a ramp-up that confirms along the way is desirable. *Table 3-9* shows the algorithm used by the hardware.

Table 3-9. Ramp-Up for Hardware-Initiated Stream

Allocation	L2 State 1	L2 State 2	Steady State	Request Type
n + 1	n + 2	n + 3	n + 4	L1
n + 2	n + 3			L2
	n + 4			L2
	n + 5	n + 6		L2
		n + 7		L2
		n + 8	n + 9	L2

Table 3-10 shows the ramp up for a software-initiated stream.

Table 3-10. Ramp-Up for Software-Initiated Stream

Allocation	Steady State	Request Type
n + 1	n + 2	L1
n + 2		L2
n + 3		L2
n + 4		L2
n + 5		L2
n + 6	n + 7	L2

The rules for PRQ management are as follows:

- Streams are allocated one at a time to the PRQ from the PFQ. The PFQ provides the stream tag number (0 to 7).
- On allocation, the real address is incremented or decremented based on the direction bit supplied. This new address is marked for scheduling to the L1. An L2 prefetch request, at the current real address ± 1, is

marked for scheduling to the L2. (During ramp-up, additional L2 prefetches can be marked for scheduling.) The stream is marked valid, and matches may occur after the stream is marked valid.

- A CAM match occurs in the prefetch queue (see *Figure 3-11* on page 119). When a demand-fetch hit[1] or miss matches an entry in the prefetch request queue, the following state occurs:

  Steady State: The real address is incremented or decremented based on the direction bit in the PRQ. The new address is scheduled for an L1 request and replaces the entry in the CAM that it matched. The new address + x^2 is scheduled for an L2 request.

  Ramp-up: If in the ramp-up state, multiple L2 requests will be scheduled. Ramp-up is achieved when the correct number of L2 requests have been requested to the L2 cache.

Scheduling L1 Prefetch Requests

The L1 prefetch request enters the LSU(0) pipeline during the ACC cycle, and subsequently enters the LMQ where normal types of LMQ checks are done. The LMQ does not check that the line prefetched is in the L1 (directory look up) but forwards the prefetch request directly to the L2. The issued operation from the instruction sequencer unit (ISU) that would have occupied this ACC cycle is rejected and sent back to the ISU. In other words, L1 prefetches have highest priority. The request is sent to the L2 through the DL0 bus.

L1 prefetches will be discarded if:

- The prefetch would be the second outstanding miss to a congruence class. The LMQ performs this check by checking bits 52:56.
- The prefetch hits a store (the line may be brought into the L1 cache, but it will not be validated).
- The prefetch hits a load (the entry will be collapsed, but the prefetch was probably unnecessary anyway).

Only one L1 prefetch will occur in any given cycle. Selection of which one of eight streams to process is based on a simple, fixed algorithm (start with 0; if it is not ready, then try 1; if it is not ready, try the next integer; and so on).

L1 prefetch requests are dropped for a window of 2 - 3 cycles when the LMQ is full. After this window, L1 prefetches are stalled until the LMQ reaches a low-water mark of four entries in the LMQ. At that time, L1 prefetches start again.

Also, the LMQ forces prefetch stalls similar to the LMQ full when the LMQ is in Next-to-Complete mode. This mode occurs when the LMQ rejects a next-to-complete load, and remains in effect until a next-to-complete load is accepted. During this time, no prefetches are accepted, and only loads that are in the next-to-complete group are accepted.

Note: The L2 will not throw away L1 prefetches due to a load hitting the L1 prefetch in the LMQ.

1. Hit in this case means the line is in the L1 cache.
2. "x" is the number of prefetches ahead being kept in the L2 cache; x can be negative if the direction is down.

Scheduling L2 Requests

L2 prefetch requests share a private bus to the L2 with translation tablewalk requests. Translation has the highest priority. The L2 will stall L2 prefetch requests when there are four such requests active in the L2.

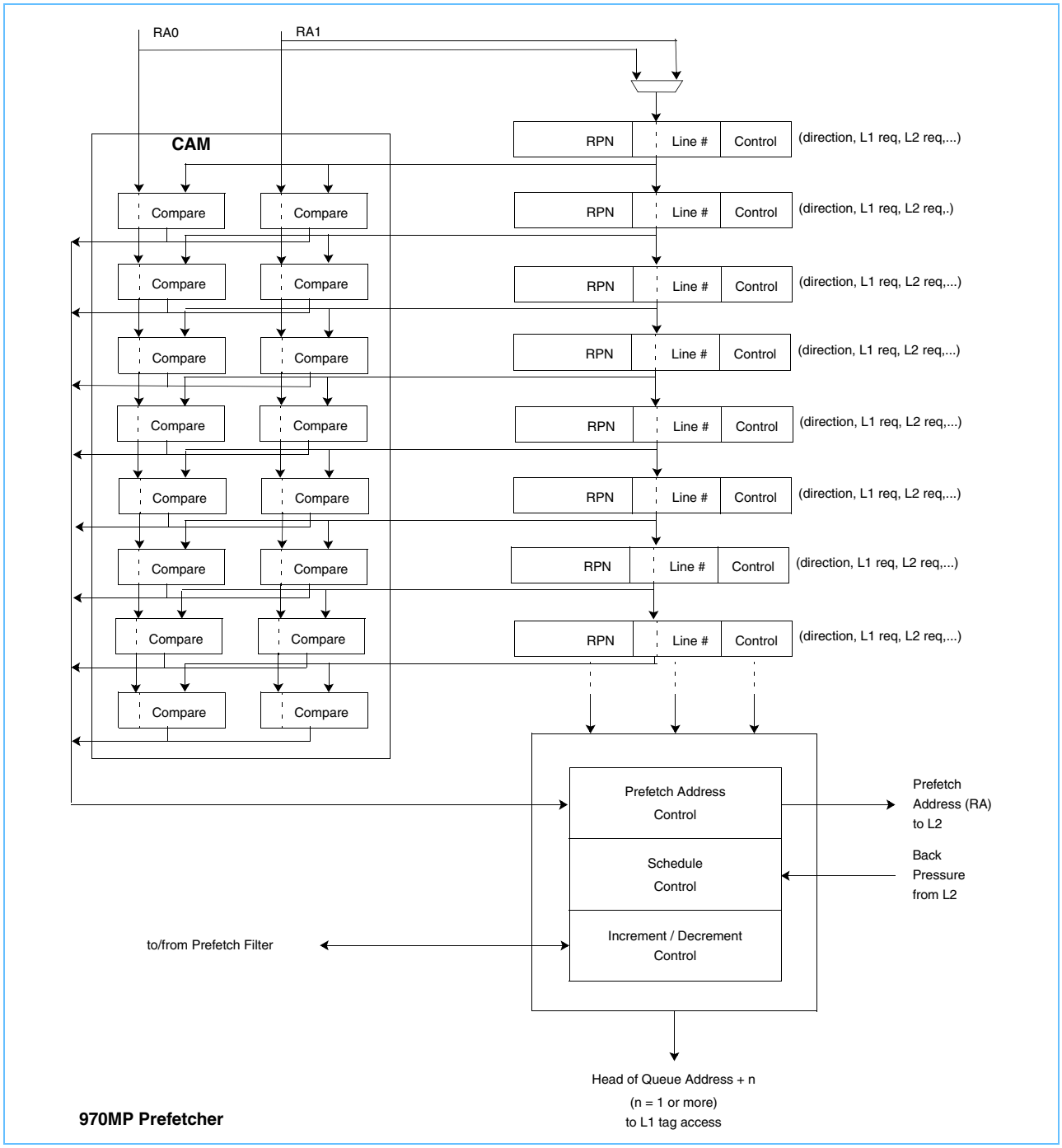
Selection of which L2 prefetch to source is a 2-step sequence:

1. Select which one of the L2 requests of a particular stream is needed for ramp-up.
2. Select one of the streams based on an algorithm similar to the one used for L1 scheduling.

Special Cases—Multiple Allocated Streams

It is possible for a stream to be allocated multiple times to the request queue. This should not happen often. When it does, all but one stream in the request queue is invalidated.

Figure 3-11. Prefetch Request Queue (PRQ)



3.5.4.3 Managing the Data Prefetch Hardware

Software can manage the data prefetch hardware by using special forms of the **dcbt** instruction. Two forms of **dcbt** variants are implemented in each 970MP processing unit.

*Optional **dcbt** Variant*

The architecture describes the first **dcbt** variant as optional. This version of the instruction includes a 2-bit Touch Hint (TH) field (instruction bits 9 - 10), which permits a program to provide a hint regarding a sequence of data cache blocks. Such a sequence is called a "data stream." A **dcbt** instruction in which TH does not equal '00' is called a "data stream variant" of **dcbt**.

Figure 3-12 shows the instruction format and interpretation of the TH field for this **dcbt** variant.

Figure 3-12. Data Cache Block Touch X-Form (Optional Variant)

dcbt RA,RB,TH

31	///	TH	RA	RB	278	/
0	6	9	11	16	21	31

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

TH Description

00 The program will probably soon load from the block containing the byte addressed by the EA.

01 The program will probably soon load from the data stream consisting of the block containing the byte addressed by the EA and an unlimited number of sequentially following blocks (that is, consisting of the blocks containing the bytes addressed by *EA* + *n* × *block_size*; where *n* equals 0, 1, 2,...).

10 Reserved

11 The program will probably soon load from the data stream consisting of the block containing the byte addressed by the EA and an unlimited number of sequentially preceding blocks (that is, consisting of the blocks containing the bytes addressed by *EA* - *n* × *block_size*; where *n* equals 0, 1, 2,...).

Restrictions

For the data stream variant cases (TH equals '01' or TH equals '11'), prefetching the stream starts even if the first block of the stream is already in the L1 data cache.

Enhanced DCBT Variant

An additional variant of the **dcbt** instruction is implemented in each 970MP processing unit. In this version, the TH field is extended to four bits (instruction bits 7 - 10) to provide the additional variant of **dcbt**. Note that the 2-bit optional variant of the software touch is a subset of the 4-bit extended version.

Figure 3-13 on page 121 provides a brief description of this variant.

Figure 3-13. Data Cache Block Touch X-Form (Enhanced Variant) (Page 1 of 2)

dcbt RA,RB,TH

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

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

TH Description

0000 The program will probably soon load from the block containing the byte addressed by the EA.

0001 The program will probably soon load from the data stream consisting of the block containing the byte addressed by the EA and an unlimited number of sequentially following blocks (that is, consisting of the blocks containing the bytes addressed by *EA* + *n* × *block_size*, where *n* equals 0, 1, 2,...).

0011 The program will probably soon load from the data stream consisting of the block containing the byte addressed by EA and an unlimited number of sequentially preceding blocks (that is, consisting of the blocks containing the bytes addressed by *EA* - *n* × *block_size*, where *n* equals 0, 1, 2,...).

1000 The **dcbt** instruction provides a hint that describes certain attributes of a data stream, and optionally indicates that the program will probably soon load from the stream. The EA, in this case, is interpreted as follows:

EA_TRUNC	D	UG	/	ID
0 56	57	58	59	60 63

Bits	Field Name	Description
0:56	EA_TRUNC	High-order 57 bits of the effective address of the first unit of the data stream. The low-order seven bits of that effective address are zero.
57	D	Direction 0 Subsequent units are the sequentially following units. 1 Subsequent units are the sequentially preceding units.
58	UG	0 No information is provided by the UG field. 1 The number of units in the data stream is unlimited, the program's need for each block of the stream is not likely to be transient, and the program will probably soon load from the stream.
59	—	Reserved
60:63	ID	Stream ID to use for this data stream.

Figure 3-13. Data Cache Block Touch X-Form (Enhanced Variant) (Page 2 of 2)

1010 The **dcbt** instruction provides a hint that describes certain attributes of a data stream, or indicates that the program will probably soon load from data streams that have been described using **dctb** instructions in which TH[0] equals '1', or will probably no longer load from such data streams.

The EA is interpreted as follows. If GO equals '1' and S ≠ '00' the hint provided by the instruction is undefined; the remainder of this instruction description assumes that this combination is not used. A completely described stream is one that has been described with both a '1000' TH values (specifying starting address and direction of the stream) and a '1010' TH value (specifying the length and transience of the stream).

Bits	Field Name	Description
0:31	—	Reserved.
32	GO	0 No information is provided by this field. 1 The program will probably soon load from all completely described streams, and will probably no longer load from any partially defined streams. All other fields of the EA are ignored.
33:34	S	00 No information is provided by this field. 01 Reserved 10 The program will probably no longer load from the data stream (if any) associated with the specified stream ID. All other fields of the EA except ID are ignored. 11 The program will probably no longer load from the data streams associated with all stream IDs. All other fields of the EA are ignored.
35:46	—	Reserved.
47:56	Unit_cnt	Number of (aligned 128 B) units in the data stream.
57	T	0 No information is provided by this field. 1 The program's need for each unit of the data stream is likely to be transient.
58	U	0 No information is provided by this field. 1 The number of units in the data stream is unlimited. The unit_cnt field is ignored.
59	—	Reserved.
60:63	ID	Stream ID.

All other TH decodes are reserved.

Restrictions

The TH equals '1000' version of the **dcbt** instruction is not recognized when MSR[DR] equals '0'.

3.5.4.4 Vector Prefetch Instruction Support

970MP Vector DST Implementation

For the 970MP processing unit, the eight hardware prefetch streams are divided into two groups of four prefetch streams. Four of these will be used for the vector prefetch instructions and will support vector-style prefetching, with some restrictions. Optionally (using a mode bit), the eight prefetch streams can be used as hardware prefetch streams with all vector data stream instructions treated as no-ops.

When four streams are configured to support vector prefetch instructions, relatively simple mapping hardware is used to convert a subset of the possible data stream touch (DST) instructions into an asynchronous prefetch of *n* cache lines into the 970MP processor caches. The first line will be placed into the L1 cache, with the remaining lines placed in the L2 cache. However, due to the nature of the DST instructions and the available data and control paths in the 970MP processing unit, these instructions will not be executed speculatively.

Map Hardware—The map hardware will map the DST BLKSIZE, BLKCOUNT, and STRIDE for the DSTs supported. The map hardware will ignore nonsupported DSTs. Positive and negative strides will be mapped.

Map Hardware Coverage—The mapping will cover cases where consecutive lines are prefetched, or where one line is prefetched at a regular multiple line stride.

When the block count is one, the number of lines prefetched will be 1 - 4 depending on the size field. When the block count is not one, there are three cases where the number of lines prefetched will be based on the stride. These are:

- Stride equals size
- Stride is 128 bytes
- Stride is less than 128 bytes

The following definitions are used in the mapping summary (some definition complexity is due to the use of zero in some **r**B fields to mean the maximum value; **r**B is the **r**B value of the DST instruction).

Size Z(0:5)	Ze || rB(3:7), where Ze = 1 if rB(3:7) = '00000'
Block Count B(0:8)	Be || rB(8:15), where Be = 1 if rB(8:15) = x'00'

Stride S(0:16)	If rB(16) = 0, then
	Stride S(0:16) = 0 || Se || rB(17:31), where Se = 1 if rB(16:31) = x'0000'
	
	If rB(16) = 1, then
	Stride S(0:16) = cb(15:31), where cb is the two's complement of rB(16) || rB(16:31)

Number of prefetch lines based on size, N_z	N_z = 4 if Z(1:4) = '0000', else Z(1:2) if Z(3:5) = '000', else Z(1:2) + 1
Number of prefetch lines based on stride, N_s	N_s = B(0:8) shifted based on the stride, as defined in *Table 3-11*

Table 3-11. Shift for N_S Based on Stride

S(7:12)	Shift B by	Direction
1 00000	2	left
0 1xxxx	1	left
0 01xxx	0	no shift
0 001xx	1	right
0 0001x	2	right
0 00001	3	right
Note: Other cases, not listed in the table, are ruled out by the mapping hardware.		

Mapping Summary

Table 3-12 shows the DST mapping summary.

Table 3-12. DST Mapping Summary

Size in Blocks	Block Count	Stride S (0:16)	Number of Prefetch Lines	Line Increment	Direction Postitve(0) Negative(1)
Z(0:5)	1	Don't Care	N_z	1	0
Z(0:5)	B not = '1'	0 000000 || Z(0:5) || 0000 OR 0 000000 || 001000 || 0000 OR 0 000000 || 000xxx || 0000 (xxx will not be 000)	N_s	1	rB(16)
don't care	B not = '1'	0 yyyyyy || yyy000 || 0000 (yyyyyy yyy not = '1')	B(0:8)	S(1:9)	rB(16)

The first row in *Table 3-12* represents the case where the block count equals one. The size of the block determines the number of lines prefetched, and the direction is, by the instruction definition, positive.

The second row represents three cases. In the first case, stride is equal to the size. The block count is shifted to give the number of lines needed based on the stride (also block size; see *Table 3-11* on page 124). In the second case, the stride is 128 bytes. In this case, the number of lines is the unshifted block count. The third case covers strides less than 128 bytes (xxx cannot be '000' by the definition of S[0:16]). Once again, the stride is used to adjust the block count to provide the number of lines to prefetch, N_s.

The third row represents the case where the stride is a multiple of the line size. Because the underlying prefetch hardware cannot handle this case, if the block size spans lines, the hardware assumes that the block size is contained in a single line. The number of lines prefetched then is the unadjusted block count; the line increment is the line multiple, S(1:9).

Because the underlying prefetch hardware does not support prefetching across a page boundary, this restriction also applies to the support for mapped vector DSTs. However, accesses of the prefetch data beyond the cache line boundary may be picked up by the hardware prefetch mechanisms, limiting the performance penalty for crossing pages.

Table 3-13. DST Mapping Examples

Number of Bytes in the Block	Stride S(0:16)	Block Count	Number of 128-Byte Lines	Line Increment	Direction Positve(0) Negative(1)
16	16	8	1	1	rB(16)
16	32	8	2	1	rB(16)
16	32	12	3	1	rB(16)
16	32	16	4	1	rB(16)
32	64	14	7	1	rB(16)
64	64	64	32	1	rB(16)
512	512	8	32	1	rB(16)
256	don't care	1	2	1	0
272	don't care	1	3	1	0
64	128	12	12	1	rB(16)
64	512	8	8	4	rB(16)

970MP Processor Vector DST Implementation Restrictions

- All DST instructions, including the stop instructions, are nonspeculative.
- DSTST is supported as a DST only.
- DSTs not falling into one of the above cases are treated as a 1-line touch.
- DSTs are terminated at a page boundary (no page crossing).
- Optionally, alignment may be handled by prefetching one line past the DST end.
- The transient hint is ignored.
- There is no monitoring of the MSR[PR] bit by the prefetch hardware.
- The prefetch hardware suspends DSTs when MSR[DR] is set to '0'.
- Negative stride cases, where the block size is greater than 128 bytes, result in incorrect prefetching of the first block. The mapping hardware fetches one line of the block and then begins the negative direction accesses.
- The total number of outstanding prefetch requests active as a result of DSTs is limited to four.
- The mapping hardware assumes strides are a power of two.

3.5.4.5 Programmability

An enable/disable for data prefetch is located in the HID4[25].

An enable for vector prefetch support is located in HID5[54:55]. These bits, if set to '00', enable the use of four of the hardware prefetch streams as vector streams. If set to '11', all eight streams are used as hardware prefetch streams, and the LSU prefetch hardware treats all vector prefetch instructions as no-ops.

4. Exceptions

The operating environment architecture (OEA) portion of the PowerPC Architecture defines the mechanism by which PowerPC processors implement exceptions (referred to as interrupts in the architecture specification). Exception conditions may be defined at other levels of the architecture. For example, the user instruction set architecture (UISA) defines conditions that may cause floating-point exceptions; the OEA defines the mechanism by which the exception is taken.

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

Although multiple exception conditions can map to a single exception vector, often a more specific condition can be determined by examining a register associated with the exception—for example, the Data Storage Interrupt Status Register (DSISR) and the Floating-Point Status and Control Register (FPSCR). The high-order bits of the Machine State Register (MSR) are also set for some exceptions. Software can explicitly enable or disable some exception conditions.

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

To prevent loss of state information, exception handlers must save the information stored in the Machine Status Save/Restore Registers, SRR0 and SRR1, soon after the exception is taken to prevent this information from being lost due to another exception being taken. Because exceptions can occur while an exception handler routine is executing, multiple exceptions can become nested. It is up to the exception handler to save the necessary state information if control is to return to the excepting program.

In many cases, after the exception handler returns, there is an attempt to execute the instruction that caused the exception (such as a page fault). Instruction execution continues until the next exception condition is encountered. Recognizing and handling exception conditions sequentially guarantees that the machine state is recoverable and processing can resume without losing instruction results.

In this book, the following terms are used to describe the stages of exception processing.

Recognition	Exception recognition occurs when the condition that can cause an exception is identified by the processor.
Taken	An exception is said to be taken when control of instruction execution is passed to the exception handler. That is, the context is saved and the instruction at the appropriate vector offset is fetched and the exception handler routine is begun in supervisor mode.
Handling	Exception handling is performed by the software linked to the appropriate vector offset. Exception handling is begun in supervisor mode (referred to as privileged state in the architecture specification).

Note: The PowerPC Architecture documentation refers to exceptions as interrupts. In this book, the term "interrupt" is reserved to refer to asynchronous exceptions and sometimes to the event that causes the exception. The PowerPC Architecture also uses the word "exception" to refer to IEEE-defined floating-point exception conditions that may cause a program exception to be taken (see the *PowerPC Microprocessor Family: The Programming Environments* manual for more information). The occurrence of these IEEE exceptions may not cause an exception to be taken. IEEE-defined exceptions are referred to as IEEE floating-point exceptions or floating-point exceptions.

Note: Previous PowerPC microprocessors supported specifying the base real address by using the exception prefix field, MSR[IP]. The 970MP microprocessor does not support this.

4.1 970MP Microprocessor Exceptions

As specified by the PowerPC Architecture, exceptions can be either precise or imprecise and either synchronous or asynchronous. Asynchronous exceptions are caused by events external to the processor's execution; synchronous exceptions are caused by instructions. The types of exceptions are shown in *Table 4-1*.

Note: All exceptions except for the maintenance exception and performance monitor exception are defined, at least to some extent, by the PowerPC Architecture.

Table 4-1. 970MP Microprocessor Exception Classifications

Synchronous/Asynchronous	Precise/Imprecise	Exception Types
Asynchronous, nonmaskable	Imprecise	Machine check, system reset
Asynchronous, maskable	Precise	External interrupt, decrementer, maintenance exception, performance monitor exception
Synchronous	Precise	Instruction-caused exceptions

These classifications are discussed in greater detail in *Section 4.2* on page 131. For a better understanding of precise exceptions, see Chapter 6, "Exceptions" of the *PowerPC Microprocessor Family: The Programming Environments* manual. Exceptions implemented in the 970MP microprocessor, and conditions that cause them, are listed in *Table 4-2 Exceptions and Conditions* on page 129.

Table 4-2. Exceptions and Conditions (Page 1 of 2)

Exception Type	Vector Offset (hex)	Causing Conditions
System reset	00100	Either the assertion of the soft reset input pin or an SCOM command sequence for "soft reset." See *Section 4.5.1 System Reset Exception* on page 138.
Machine check	00200	There are many causes of a machine check exception. See *Section 4.5.2 Machine Check Exceptions* on page 139.
Data storage	00300	Page fault, as defined in the PowerPC Architecture. See *Section 4.5.3 Data Storage Exception* on page 141.
Data segment	00380	Data segment fault, as defined in the PowerPC AS Architecture. See *Section 4.5.4 Data Segment Exception* on page 141.
Instruction storage	00400	Page fault, as defined in the PowerPC Architecture. See *Section 4.5.5 Instruction Storage Exception* on page 141.
Instruction segment	00480	Instruction segment fault, as defined in the PowerPC AS Architecture. See *Section 4.5.6 Instruction Segment Exception* on page 141.
External interrupt	00500	Assertion of the external interrupt input signal. See *Section 4.5.7 External Interrupt Exception* on page 142.
Alignment	00600	There are many causes of an alignment exception. See *Section 4.5.8 Alignment Exception* on page 142.
Program	00700	As defined by the PowerPC Architecture (for example, an instruction opcode error). See *Section 4.5.9 Program Exception* on page 142.
Floating-point unavailable	00800	As defined by the PowerPC Architecture. See *Section 4.5.10 Floating-Point Unavailable Exception* on page 143.
Decrementer	00900	As defined by the PowerPC Architecture. When the most-significant bit of the Decrementer Register (DEC) changes to '1' and MSR[EE] equals '1', it is the responsibility of the service routine for the decrementer exception to clear DEC[0]. See *Section 4.5.11 Decrementer Exception* on page 143.
Hypervisor decrementer	00980	The Hypervisor Decrementer is similar to the decrementer and is used to return control to the hypervisor. This interrupt is activated when no higher priority interrupt is active and MSR[EE]='1' or MSR[HV]='0' and the Hypervisor Decrementer is negative (HDEC[0]='1'). This is a level sensitive interrupt and as such it is the responsibility of the interrupt service routine to clear HDEC[0].
System call	00C00	Execution of the System Call (**sc**) instruction. See *Section 4.5.12 System Call Exception* on page 143.
Trace	00D00	MSR[SE] equals '1' or MSR[BE] equals '1', and a trace-marked instruction successfully completes. See *Section 4.5.13 Trace Exception* on page 143.
Performance monitor	00F00	The MSR[EE] bit is set, the MMCR0[PMXE] bit is set, and any of the performance monitor counters overflow. The performance monitor exception can also be triggered by the '0' to '1' transition of a particular time-base bit. See *Section 4.5.14 Performance Monitor Exception* on page 144.
VPU unavailable	00F20	No higher priority exception exists, an attempt is made to execute a vector instruction, and MSR[VP] equals '0'. See *Section 4.5.15 VPU Unavailable Exception* on page 144.

Table 4-2. Exceptions and Conditions (Page 2 of 2)

Exception Type	Vector Offset (hex)	Causing Conditions
Instruction address breakpoint	01300	PowerPC 970MP microprocessor does not support a visible form of the instruction address breakpoint facility. The instruction address breakpoint feature is accessible through the support processor interface. See *Section 4.5.16 Instruction Address Breakpoint Exception* on page 145.
Maintenance	01600	This exception can be signaled by a number of internal events, as well as by explicit commands from the support processor. See *Section 4.5.17 Maintenance Exception* on page 145.
VPU assist	01700	This exception occurs when operating in Java mode and the input operands or the result of an operation are denormalized. See *Section 4.5.18 VPU Assist Exception* on page 145.

4.2 Exception Recognition and Priorities

Exceptions are roughly prioritized by exception class, as follows.

- Nonmaskable, asynchronous exceptions have priority over all other exceptions. These are system reset and machine check exceptions. These exceptions cannot be delayed and do not wait for completion of any precise exception handling. (However, the machine check exception condition can be disabled so the condition causes the processor to go directly into the checkstop[1] state).
- Synchronous, precise exceptions are caused by instructions and are taken in strict program order.
- Imprecise exceptions (imprecise mode floating-point enabled exceptions) are caused by instructions, and they are delayed until higher priority exceptions are taken.

  Note: The 970MP microprocessor does not implement an exception of this type.
- Maskable asynchronous exceptions (external, decrementer, maintenance, performance monitor, and exceptions) are delayed if higher priority exceptions are taken.

Section 4.3 Exception Processing on page 133 describes how the 970MP microprocessor handles exceptions up to the point of signalling the appropriate interrupt to occur. Note that a recoverable state is reached if the completed store queue is empty (drained, not cancelled) and any instruction that is next in program order and has been signaled to complete has completed. If MSR[RI] equals '0', the 970MP processing unit is in a nonrecoverable state. Also, instruction completion is defined as updating all architectural registers associated with that instruction, and then removing that instruction from the completion buffer.

4.2.1 Exception Priorities

The following list is a summary of the exception priorities for the 970MP microprocessor:

1. System reset exception
2. Machine check exception
3. Instruction dependent (as follows)
   - Fixed-point loads and stores
     - Mode dependent loads and stores
       (1) Illegal instruction type of program exception
       (2) Privileged type of program exception (for example, MSR[PR] set to'1')
     - Data segment exception
     - Data storage exception
     - Alignment exception
     - Trace exception
   - Floating-point loads and stores
     - Floating-point unavailable exception
     - Data segment exception
     - Data storage exception (DSI)
     - Alignment exception
     - Trace exception
   - Other floating-point instructions
     - Floating-point unavailable exception

1. Hardware has detected a condition that it cannot resolve, and which prevents normal operation. It stops executing instructions, responding to interrupts, and so on.

  - Precise-mode, floating-point-enabled, exceptions type of program exception
  - Trace exception
- Vector instructions
  - VPU unavailable exception
  - Trace exception
- Return from Exception Doubleword (**rfid**) instruction, Move to Machine State Register (**mtmsr**), Move to Machine State Register Doubleword (**mtmsrd**)
  - Precise-mode, floating-point-enabled, exceptions type of program exception
  - Trace exception (for **mtmsr** or **mtmsrd** only)
- Other instructions
  - Exceptions that are mutually exclusive and the same priority:
    (1) Trap type of program exception
    (2) System call
    (3) Privileged instruction type of program exception
    (4) Illegal instruction type of program exception
  - Trace exception
  - VPU assist exception
- Instruction segment exception
- Instruction storage exception

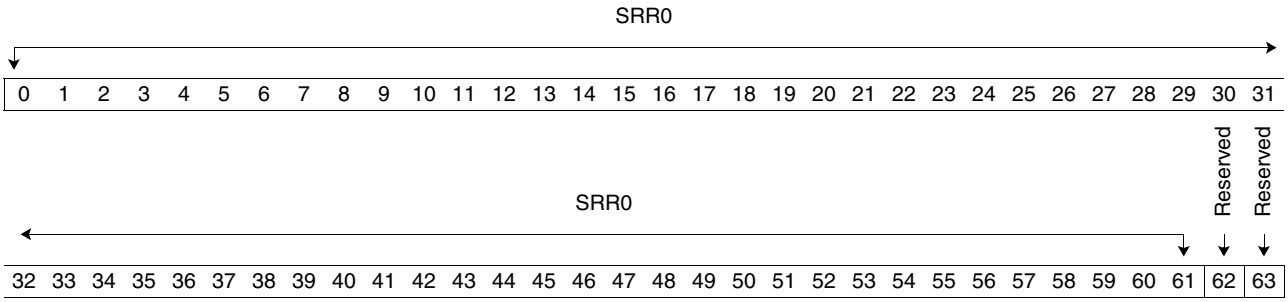
4. Maintenance exception
5. External interrupt
6. Performance monitor exception
7. Decrementer exception

4.3 Exception Processing

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

4.3.1 Machine Status Save/Restore Register 0 (SRR0)

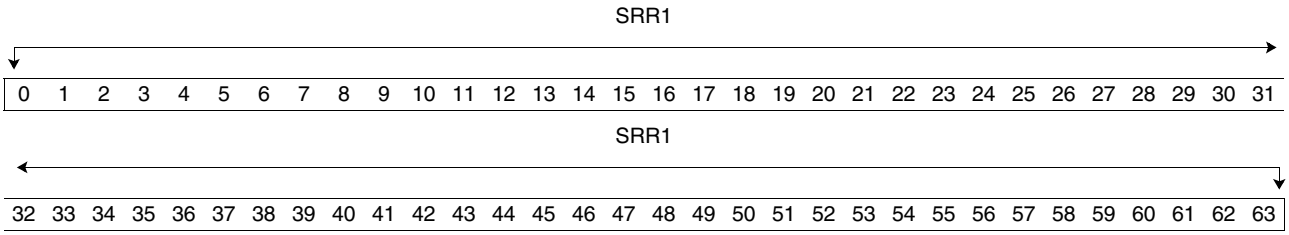
When an exception occurs, the address saved in SRR0 determines where instruction processing should resume when the exception handler returns control to the interrupted process. Depending on the exception, this may be the address in SRR0 or at the next address in the program flow. All instructions in the program flow preceding this one will have completed execution and no subsequent instruction will have begun execution. This may be the address of the instruction that caused the exception or the next one (as in the case of a system call, trace, or trap exception). The SRR0 Register is shown below.



Bits	Field Name	Description
0:61	SRR0	Holds the effective address (EA) for the instruction in the interrupted program flow.
62	—	Reserved. Returns a zero when read.
63	—	Reserved. Returns a zero when read.

4.3.2 Machine Status Save/Restore Register 1 (SRR1)

SRR1 is used to save machine status (selected MSR bits and possibly other status bits as well) on exceptions and to restore those values when an **rfid** instruction is executed. SRR1 is shown below.

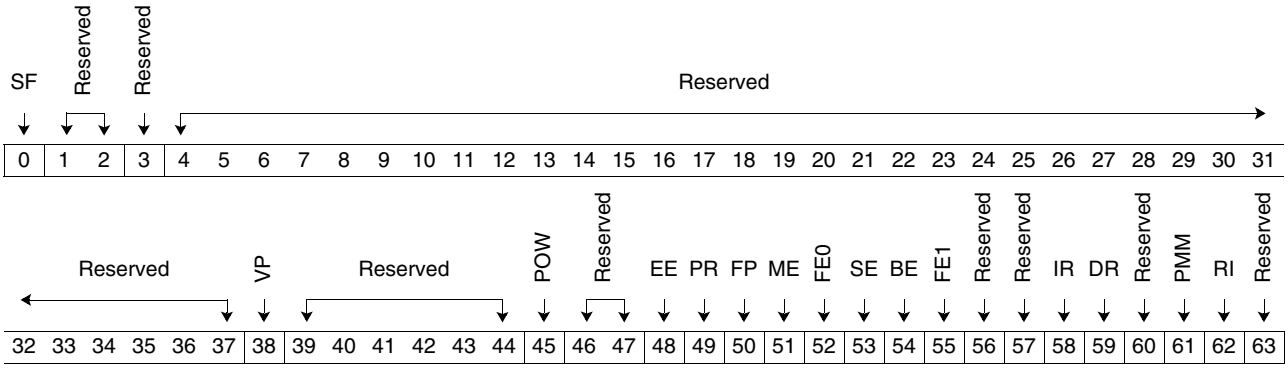


Bits	Field Name	Description
0:63	SRR1	Exception-Specific Information and MSR Bit Values For most exceptions, bits 33 - 36 and 42 - 47 of SRR1 are loaded with exception-specific information. Bits 0 - 32, 37 - 41, and 48 - 63 of SRR1 are loaded with a copy of the corresponding bits of the MSR (before taking the exception).

Note: The function of the SRR1 is to save the current state of the machine (that is, the MSR) before a temporary state is invoked to service exceptions. After the servicing of the exception, the contents of SRR1 are returned to the MSR and the code stream can continue.

4.3.3 Machine State Register (MSR)

The format of the 970MP processing unit's MSR is below.



Bits	Name	Description
0	SF	64-bit mode. 0 Processor runs in 32-bit mode. 1 Default mode. Processor runs in 64-bit mode.
1:2	—	Reserved. Returns zeros when read.
3	HV	Hypervisor mode. Set when running on a non-partitioned system or when "hypervisor code" is executing on a partitioned system. MSR[HV] can be set to '1' only by the system call instruction and some interrupts. It can be set to '0' only by the **rfid** and **hrfid** instructions.
4:37	—	Reserved. Returns zeros when read.
38	VP	Vector processor available. 0 The processor prevents execution of all vector instructions, including loads, stores, and moves. If such execution is attempted, a VPU unavailable exception is raised. 1 The processor can execute all vector instructions. The Vector Save/Restore Register (VRSAVE) is not protected by MSR[VP]. None of the data streaming family of instructions (**dst**, **dstt**, **dstst**, **dststt**, **dss**, and **dssall**) are affected by MSR[VP].
39:44	—	Reserved. Returns zeros when read.
45	POW	Power-management enable 0 Power management disabled (normal operation mode). 1 Power management enabled (reduced power mode).
46:47	—	Reserved. Returns zeros when read.
48	EE	External exception enable. 0 The processor delays recognition of external exceptions and decrementer exception conditions. 1 The processor is enabled to take an external exception or the decrementer exception. **Note:** Setting MSR[EE] masks not only the architecture-defined external exception and decrementer exceptions, but also the 970MP-specific debug and performance monitor exceptions.
49	PR	Problem state (user mode). 0 The processor is privileged to execute any instruction. 1 The processor can only execute nonprivileged instructions.
50	FP	Floating-point available. 0 The processor prevents dispatch of floating-point instructions, including floating-point loads, stores, and moves. 1 The processor can execute floating-point instructions and can take floating-point enabled program exceptions.

Bits	Name	Description
51	ME	Machine check enable. 0 Machine check exceptions are disabled. If one occurs, the system enters checkstop. 1 Machine check exceptions are enabled. Only **rfid** instructions can alter MSR[ME].
52	FE0	IEEE floating-point exception mode 0.
53	SE	Single-step trace enable. 0 The processor executes instructions normally. 1 The processor generates a single-step trace exception upon the successful execution of every instruction except **rfid,** Instruction Cache Synchronize (**isync**), and **sc**. Successful execution means that the instruction caused no other exception.
54	BE	Branch trace enable. 0 The processor executes branch instructions normally. 1 The processor generates a branch type of trace exception when a branch instruction executes successfully.
55	FE1	IEEE floating-point exception mode 1.
56	—	Reserved. Returns a zero when read.
57	—	Reserved. Returns a zero when read.
58	IR	Instruction address translation. 0 Instruction address translation is disabled. 1 Instruction address translation is enabled.
59	DR	Data address translation 0 Data address translation is disabled. If data stream touch (**dst**) and data stream touch for store (**dstst**) instructions are executed when DR equals '0', the results are boundedly undefined. 1 Data address translation is enabled. Data stream touch (**dst**) and data stream touch for store (**dstst**) instructions are supported when DR equals '1'.
60	—	Reserved. Returns a zero when read.
61	PMM	Performance monitor mode. This register bit is used to enable and disable performance monitor activity controlled by the process mark bit.
62	RI	Indicates whether a system reset or machine check exception is recoverable. 0 Exception is not recoverable. 1 Exception is recoverable. The RI bit indicates whether, from the perspective of the processor, it is safe to continue (that is, processor state data such as that saved to SRR0 is valid), but it does not guarantee that the interrupted process is recoverable. Exception handlers must look at SRR1[RI] to determine this.
63	—	Reserved. Returns a zero when read.

The 970MP microprocessor provides precise floating-point exceptions whenever either of the floating-point enabled exception mode bits (MSR[FE0] and MSR[FE1]) are set. *Table 4-3 IEEE Floating-Point Exception Mode Bits* on page 136 explains how the bits are used to set the mode. In all cases, the 970MP processing unit aggressively executes the floating-point instructions (even out-of-order as required), and sorts out any resulting exceptions at completion time. In some cases, due to the group-oriented instruction tracking scheme used by the 970MP microprocessor, when an exception is detected, the hardware will flush the pipeline and re-dispatch the instructions individually in order to provide the precise exception. Since this only happens if an exception is to be taken, it does not represent a measurable decrease in performance.

Table 4-3. IEEE Floating-Point Exception Mode Bits

FE0	FE1	Mode
0	0	Floating-point exceptions disabled.
0	1	Imprecise nonrecoverable. For this setting, the 970MP microprocessor operates in floating-point precise mode.
1	0	Imprecise recoverable. For this setting, the 970MP microprocessor operates in floating-point precise mode.
1	1	Floating-point precise mode.

4.3.4 Enabling and Disabling Exceptions

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

- IEEE floating-point enabled exceptions (a type of program exception) are ignored when both MSR[FE0] and MSR[FE1] are cleared. If either bit is set, all IEEE enabled floating-point exceptions are taken and cause a program exception.
- Asynchronous, maskable exceptions (external, decrementer, performance monitor, and maintenance exceptions) are enabled by setting MSR[EE]. When MSR[EE] equals '0', recognition of these exception conditions is delayed. MSR[EE] is cleared automatically when an exception is taken to delay recognition of conditions causing those exceptions.
- A machine check exception can occur only if the machine check enable bit, MSR[ME], is set. If MSR[ME] is cleared, the processor goes directly into checkstop state when a machine check exception condition occurs.
- System reset exceptions cannot be masked.

4.3.5 Exception Processing Steps

After it is determined that the exception can be taken (by confirming that any instruction-caused exceptions occurring earlier in the instruction stream have been handled, and by confirming that the exception is enabled for the exception condition), the processor does the following steps:

1. Loads SRR0 with an instruction address that depends on the type of exception. Normally, this is the instruction that would have completed next had the exception not been taken. See the individual exception description (*Section 4.5* beginning on page 138) for details about how this register is used for specific exceptions.
2. Loads SRR1[33:36, 42:47] with information specific to the exception type.
3. Loads SRR1[0:32, 37:41, 48:63] with a copy of the corresponding MSR bits (prior to the exception).
4. Sets the MSR as described in *Section 4.5 Exception Definitions* on page 138. The new values take effect as the first instruction of the exception-handler routine is fetched.

   Note: MSR[IR] and MSR[DR] are cleared for all exception types. Therefore, address translation is disabled for both instruction fetches and data accesses beginning with the first instruction of the exception-handler routine.

Instruction fetch and execution resumes, using the new MSR value, at a location specific to the exception type. The location is determined by adding the exception's vector offset (see *Table 4-2* on page 129) to the value in the Hardware Interrupt Offset Register (HIOR). For a machine check exception that occurs when MSR[ME] equals '0' (machine check exceptions are disabled), the checkstop state is entered (the machine stops executing instructions).

4.3.6 Setting the Recoverable Exception in the MSR

The recoverable exception (RI) bit in the MSR was designed to indicate to the exception handler whether the exception is recoverable. When an exception occurs, the RI bit is copied from the MSR to SRR1 and cleared in the MSR. All exceptions are disabled except machine check. If a machine check exception occurs while MSR[RI] is clear, a '0' value is found in SRR1[RI] to indicate that the machine state is definitely not recoverable. When MSR[RI] equals '1', the exception is recoverable as far as the current state of the machine and all programs concerned including noncritical machine checks. Thus, in all exceptions, if SRR1[RI] is cleared, the machine state is not recoverable. If it is set, the exception is recoverable with respect to the processor and all programs. An operating system can handle MSR[RI] as follows:

- Use the Special Purpose Registers (SPRG0-SPRG3) to aid in saving the machine state. IBM suggests pointing SPRG0 to a stack save area in memory and saving three General Purpose Registers (GPRs) in SPRG1-3. Move SPRG0 into one of the GPRs that was saved. This GPR now points to the save area in memory. Move the GPRs, SRR0, SRR1, SPRG1-3, and other registers to be used by the exception routine into the stack save area. Update SPGR0 to point to a new save area. Set MSR[RI] to indicate that machine state has been saved. Also set MSR[EE] if you want to re-enable external exceptions.
- When exception processing is complete, clear MSR[EE] and MSR[RI]. Adjust SPRG0 to point to the stack saved area, restore the GPRs, SRR0 and SRR1, and any other register that you may have saved, execute **rfid**. This returns the processor to the interrupted program.

4.3.7 Returning from an Exception Handler

The **rfid** instruction performs context synchronization by allowing previously-issued instructions to complete before returning to the interrupted process. In general, execution of the **rfid** instruction ensures the following:

- All previous instructions have completed to a point where they can no longer cause an exception.
- Previous instructions complete execution in the context (privilege, protection, and address translation) under which they were issued.
- The **rfid** instruction copies SRR1 bits back into the MSR, and resets the MSR[POW] bit.
- Instructions fetched after this instruction execute in the context established by this instruction.
- Program execution resumes at the instruction indicated by SRR0.

For a complete description of context synchronization, see Chapter 6, "Exceptions" of the *PowerPC Microprocessor Family: The Programming Environments* manual.

4.4 Process Switching

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

- The Synchronize (**sync**) instruction orders the effects of instruction execution. All instructions previously initiated appear to have completed before the **sync** instruction completes, and no subsequent instructions appear to be initiated until the **sync** instruction completes.
- The Instruction Cache Synchronize (**isync**) instruction waits for all previous instructions to complete and then discards any fetched instructions, causing subsequent instructions to be fetched (or refetched) from memory and to execute in the context (privilege, translation, and protection) established by the previous instructions.
- The Store Word Conditional Indexed/Store Doubleword Conditional Indexed (**stwcx./stdcx.**) instruction clears any outstanding reservations, ensuring that a Load Word and Reserve Indexed/Load Double Word and Reserve Indexed (**lwarx/ldarx)** instruction in an old process is not paired with an **stwcx./stdcx.** instruction in a new one.

The operating system should set MSR[RI] as described in *Section 4.3.6 Setting the Recoverable Exception in the MSR* on page 137.

4.5 Exception Definitions

When an exception/interrupt is taken, all bits in the MSR are set to '0', with the following exceptions:

- Exceptions always set MSR[SF] to '1'.
- Only the machine check exception sets MSR[ME] to '0'. All other exceptions leave MSR[ME] unchanged.

The following sections describe the implementation-dependent aspects of the exceptions.

Note: If a description is not provided, the 970MP microprocessor behaves as described in the *PowerPC Architecture books*.

4.5.1 System Reset Exception

The system reset exception is a non-maskable, asynchronous exception that is caused by the assertion of either the soft reset input pin, or by the SCOM command sequence for soft reset.

The Not Hard Reset bit in HID0[15] can be used to help software distinguish between a hard reset and a soft reset. To use this capability, software should initially set this bit to a '1'. Later, when a system reset exception is taken, software can check the state of this bit to determine which type of reset occurred. If the bit is still set, then the reset was a soft reset, and if the bit is a zero, the reset was a hard reset.

4.5.2 Machine Check Exceptions

The are several possible causes of machine check exceptions in the 970MP microprocessor, some of which are generally recoverable, and some of which are non-recoverable.

The following causes of machine check exceptions are precise and synchronous with the instruction that caused the operation that encountered the error (that is, SRR0 contains the address of the instruction that caused the operation).

- The detection of a parity error in the L1 data cache (D-cache), the L1 D-cache tag, the data effective-to-real-address translation (D-ERAT), the translation lookaside buffer (TLB), or the segment lookaside buffer (SLB) during the execution of a load or store instruction. If the exception is caused by a soft error, then executing the appropriate sequence of instructions in the machine check handler program will clear the error condition without causing any loss of state, permitting the interrupted program to resume if MSR[RI] was a '1' when the instruction that encountered the error was executed.

  Note: The L1 D-cache and the L1 D-cache tag parity errors are recovered by hardware in the 970MP processing unit (default mode), without a machine check interrupt.
- The detection of an uncorrectable error checking and correction (ECC) error in the L2 cache when a load instruction is executed.
- The detection of an uncorrectable ECC error in the L2 cache while the page table is being searched in the process of translating an address.
- The detection of erroneous data that is being returned to satisfy a load instruction for which the effective address specified a location in caching inhibited memory.

For hard errors, these characteristics cannot be reliably provided on a machine check, because it is likely that the failure will prevent reliable execution. Additionally, a machine check exception that occurs when MSR[ME] equals '0' results in a checkstop.

In addition, there are a few possible sources for asynchronous machine check exceptions. A machine check exception is taken when the machine check input pin is asserted, if enabled by setting HID0[32] to '1'. The Fault Isolation Register (FIR), debug logic, and hang recovery logic can also be programmed to induce machine check exceptions for various error conditions. Since these signals are asynchronous with respect to the executing program, asynchronous machine checks may or may not be recoverable. Software can use the MSR[RI] bit to help identify the cases where the machine check exception is recoverable.

Information about the suspected source of the error condition is logged into either the SRR1 Register, the DSISR Register, or both as defined in *Table 4-4* on page 140 for synchronous machine checks.

Table 4-4. Register Settings for Machine Check Exception

Register	Bits	Setting
SRR0	0:63	Effective address of the next instruction that would have executed if the machine check exception was not taken. When this is a recoverable machine check due to a load that has surfaced an error, this will be the address of the load instruction itself (the 970MP microprocessor allows the instruction to execute to surface the error, but inhibits the commitment of the results). When this is a recoverable machine check due to an instruction fetch surfacing an error, this will be the address of an instruction that initiated the memory/cache access.
SRR1	0:41	Loaded from MSR.
	42	Exception caused by instruction fetch unit (IFU) detection of a hardware uncorrectable error (UE)
	43	Exception caused by load/store detection of error (see DSISR below)
	44:45	Exception cause indicated by the following encoding: 00 No error encoded. 01 Exception caused by an SLB parity error detected while translating an instruction fetch address. 10 Exception caused by a TLB parity error detected while translating an instruction fetch address. 11 Exception caused by a hardware uncorrectable error (UE) detected while doing a reload of an instruction-fetch TLB tablewalk.
	46:61	Loaded from MSR.
	62	Loaded from MSR[62] if recoverable. Otherwise, set to zero.
	63	Loaded from MSR.
DSISR	0:15	All zeros.
	16	Exception caused by a UE deferred error (the Data Address Register [DAR] is undefined).
	17	Exception caused by a UE deferred error during a tablewalk (D-side).
	18	Exception was caused by a software-recoverable parity error in the L1 D-cache.
	19	Exception was caused by a software-recoverable parity error in the L1 D-cache tag.
	20	Exception was caused by a software-recoverable parity error in the D-ERAT.
	21	Exception was caused by a software-recoverable parity error in the TLB.
	22	Zero
	23	Exception was caused by an SLB parity error (may not be recoverable). This condition could occur if the effective segment ID (ESID) fields of two or more SLB entries contain the same value.
	24:31	All zeros
DAR	0:63	Effective address computed by a load or store instruction that caused the operation that encountered a parity error in the D-ERAT, TLB, or SLB, or that encountered an uncorrectable error while attempting to reload a TLB entry. Effective address computed by the load instruction that caused the operation that encountered a parity error in the L1 D-cache or L1 D-cache tag arrays For all other types of machine check exceptions, the DAR is undefined (including when the operand of the load instruction contains a UE).

Note: As mentioned above, the machine check exception handler is expected to help hardware recover from certain types of D-cache, D-cache directory, D-ERAT, and TLB errors detected by the hardware. In general terms, the exception handler should:

- Check whether the machine check exception is recoverable by looking at the state of the RI bit in SRR1.
- Determine the type of error that caused the machine check by looking at the state of the SRR1 and DSISR Registers.
- Flush the contents of the array that reported the detected error (this process is slightly different for each of the possible arrays).
- Return to the interrupted process.

If no error is encoded in SRR1[44:45], then the exception is likely caused by an asynchronous machine check, in which case the exception handler should access the Asynchronous Machine Check Register through the SCOMC facility.

4.5.3 Data Storage Exception

The 970MP microprocessor implements the data storage exception as described in the PowerPC Architecture (OEA). A DSI exception occurs when no higher priority exception exists and an error condition related to a data memory access occurs. In case of a TLB miss for a load, store, or cache operation, a DSI exception is taken if the resulting hardware table search causes a page fault.

When this exception is taken, execution resumes at effective address x'00300'.

4.5.4 Data Segment Exception

The 970MP microprocessor implements the data segment exception as described in the PowerPC Architecture (OEA). A data segment exception occurs when no higher priority exception exists and a data access cannot be performed because data address translation is enabled (MSR[DR] is '1') and the effective address of any byte of the storage location specified by a Load, Store, Instruction Cache Block Invalidate (**icbi**), Data Cache Block Set to Zero (**dcbz**), Data Cache Block Store (**dcbst**), Data Cache Block Flush (**dcbf**), External Control In Word Indexed (**eciwx**), or External Control Out Word Indexed (**ecowx**) instruction cannot be translated to a virtual address.

When this exception is taken, execution resumes at effective address x'00380'.

4.5.5 Instruction Storage Exception

The 970MP microprocessor implements the instruction storage exception as described in the PowerPC Architecture (OEA). An instruction storage interrupt (ISI) exception occurs when no higher priority exception exists and an attempt to fetch the next instruction fails.

When this exception is taken, execution resumes at effective address x'00400'.

4.5.6 Instruction Segment Exception

The 970MP microprocessor implements the instruction segment exception as described in the PowerPC Architecture (OEA). An instruction segment exception occurs when no higher priority exception exists and next instruction to be executed cannot be fetched because instruction address translation is enabled (MSR[IR] is '1') and the effective address cannot be translated to a virtual address.

When this exception is taken, execution resumes at effective address x'00480'.

4.5.7 External Interrupt Exception

In the 970MP microprocessor, an external interrupt is signaled by the assertion of the external interrupt input signal. The external interrupt signal is expected to remain asserted until the processor has actually taken the interrupt (failure to meet this requirement may lead the processor to not recognize the interrupt request).

4.5.8 Alignment Exception

An alignment exception is taken if any of the following conditions are detected:

- **lwarx**, **stwcx,** Load Multiple Word (**lmw**)**,** Store Multiple Word (**stmw**) instructions with non-word aligned addresses
- **ldarx** and **stdcx** instructions with non-double word aligned addresses
- **lmw** and **stmw** instructions to storage marked cache-inhibited
- Load String Word Immediate (**lswi**), Load String Word Indexed (**lswx**), Store Sting Word Immediate (**stswi**), and Store String Word Indexed (**stswx**) instructions to storage marked cache-inhibited
- **dcbz** to storage marked cache-inhibited (a **dcbz** to cache-inhibited space is treated as a no-op instead of causing an alignment interrupt if the dcbz_ieq1_align bit in the mode ring is set to a '0')
- Any load or store to storage marked cache-inhibited that is not naturally aligned
- Floating-point load single instructions that are not word aligned and cross a 32-byte boundary
- Floating-point store instructions that are not word aligned and cross a 4-KB boundary
- When HID4[24] is set, some forms of unaligned storage accesses that are normally handled by the hardware are forced to take an alignment exception (to assist in debugging).

Table 4-5. Register Settings for Alignment Exception

Register	Bits	Setting
DSISR	0:31	Unchanged
DAR	0:63	Set to the effective address computed by the load or store instruction that caused the alignment exception.When the exception is caused by an unsupported access to cache-inhibited space, the DAR will be set to the effective address of the first access into the cache-inhibited space.

4.5.9 Program Exception

The 970MP microprocessor implements the program exception as it is defined by the PowerPC Architecture (OEA). A program exception occurs when no higher priority exception exists and one or more of the exception conditions defined in the OEA occur.

The 970MP microprocessor invokes the program exception for a system illegal instruction when it detects any instruction from the illegal instruction class. The 970MP processing unit fully decodes the special purpose register (SPR) field of the instruction. If an undefined SPR is specified, a program exception is taken.

When this exception is taken, execution resumes at effective address x'00700'.

4.5.10 Floating-Point Unavailable Exception

The floating-point unavailable exception is implemented as defined in the PowerPC Architecture. When a floating-point unavailable exception is taken, instruction fetching resumes at the location determined by adding the offset x'00800' to the HIOR value.

4.5.11 Decrementer Exception

The decrementer exception is implemented as defined in the PowerPC Architecture. A decrementer exception occurs when no higher priority exception exists, the decrementer is negative (DEC[0] equals '1'), and MSR[EE] equals '1'. The decrementer exception is level sensitive. It is the responsibility of the interrupt service routine to clear DEC[0].

When this exception is taken, execution resumes at effective address x'0000_0000_0000_0900'.

4.5.12 System Call Exception

The 970MP microprocessor implements the system call exception as described in the PowerPC Architecture (OEA). A system call exception occurs when a system call (**sc**) instruction is executed.

When this exception is taken, execution resumes at effective address x'00C00'.

4.5.13 Trace Exception

The trace exception is taken when the single-step trace enable bit (MSR[SE]) or the branch trace enable bit (MSR[BE]) is set and an instruction successfully completes. After a trace exception is taken, SRR0, SRR1, Sampled Instruction Address Register (SIAR), and Sampled Data Address Register (SDAR) are set as shown in *Table 4-6*.

Table 4-6. Register Settings for Trace Exception

Register	Bits	Setting
SRR0	0:63	Set as specified in the architecture.
SRR1	0:32	Loaded from the MSR.
	33:34	'10'
	35	Set for a load instruction; otherwise, cleared. Not set for a zero-length **lswx** instruction.
	36	Set for a store instruction; otherwise, cleared. Not set for a zero-length **stswx** instruction.
	37:41	Loaded from the MSR.
	42	Set for a **lwarx/ldarx** or **stwcx/stdcx** instruction; otherwise, cleared.
	43	Set to '1'.
	44	Set to '0'.
	45:47	Set to '0'.
	48:63	Loaded from the MSR.
SIAR	0:63	Set to the effective address of the traced instruction.
SDAR	0:63	If the instruction that took the trace interrupt was a storage access instruction, the SDAR is set to the effective address of the storage access. SDAR is not set if an X-form Load String or Store String instruction specifies an operand length of zero.

If either MSR bits SE or BE is set to '1' by a Return from Interrupt or Move to MSR instruction, the contents of SIAR and SDAR are undefined until a trace interrupt occurs.

4.5.14 Performance Monitor Exception

The performance monitor exception is signalled when the MSR[EE] bit is set, and a performance monitor exception condition occurs. See *Chapter 10 970MP Performance Monitor* for a description of performance monitor exception conditions.

The following registers are set when a performance monitor exception occurs.

Table 4-7. Register Settings for the Performance Monitor Exception

Register	Bits	Setting
SRR0	0:63	Set to the effective address of the instruction that the processor would have attempted to execute next if no exception conditions were present.
SRR1	0:32	Loaded from the MSR.
	33	Set to '1' if the contents of the SDAR and the SIAR are associated with the same instruction.
	34:63	Loaded from the MSR.
SIAR	0:63	Set to the effective address of the marked instruction, where the marked instruction is an instruction that was executing, possibly out-of-order, at or around the time that the performance monitor exception occurred. The contents of the SIAR may be altered by the processor if and only if MMCR0[PMEE] equals '1'. Thus, after a performance monitor exception occurs, the contents of SIAR are not altered by the processor until software sets MMCR0[PMEE] to '1'. After software sets MMCR0[PMEE] to '1', the contents of SIAR are undefined until the next performance monitor exception occurs.
SDAR	0:63	Set to the effective address of the storage operand of an instruction that was executing, possibly out-of-order, at or around the time that the performance monitor exception occurred. This storage operand is called the marked data and may be, but need not be, the storage operand (if any) of the marked instruction. If the performance monitor exception causes a performance monitor interrupt, SRR1 indicates whether the marked data is in fact the storage operand of the marked instruction. The contents of the SDAR may be altered by the processor if and only if MMCR0[PMEE] equals '1'. Thus, after a performance monitor exception occurs, the contents of SDAR are not altered by the processor until software sets MMCR0[PMEE to '1'. After software sets MMCR0[PMEE] to '1', the contents of SDAR are undefined until the next performance monitor exception occurs.

4.5.15 VPU Unavailable Exception

This exception occurs if there is an attempt to execute any vector instruction, including a vector load or store, with MSR[VP] negated. After this interrupt, execution resumes at offset x'0000_0000_0000_0F20'. The register settings for this interrupt are shown in *Table 4-8*.

Note: A **mtspr** or **mfspr** instruction that references the VRSAVE Register will not cause this interrupt.

Table 4-8. Register Settings for VPU Unavailable Interrupt

Register	Bits	Setting
SRR0	0:63	Set to the effective address of the instruction that caused the interrupt.
SRR1	0:32	Loaded from the MSR.
	33:36	Set to zeros.
	37:41	Loaded from the MSR.
	42:47	Set to zeros.
	48:63	Loaded from the MSR.

4.5.16 Instruction Address Breakpoint Exception

The 970MP microprocessor does not support a visible form of the instruction address breakpoint facility. The instruction address breakpoint feature is accessible through the support processor interface.

When this exception is taken, execution resumes at effective address x'01300'.

4.5.17 Maintenance Exception

The 970MP microprocessor provides support for an implementation-dependent maintenance exception. This exception can be signaled by a number of internal events, as well as by explicit commands from the support processor.

When this exception is taken, execution resumes at effective address x'0000_0000_0000_1600'.

This exception is controlled by the MSR[EE] bit in a manner similar to external interrupts. The register settings for this exception are shown in *Table 4-9* on page 145.

Table 4-9. Register Settings for Maintenance Exception

Register	Bits	Setting
SRR0	0:63	Set to the effective address of the next instruction that would have executed had the exception not been taken.
SRR1	0:32	Loaded from the MSR.
	33:36	Set to zeros.
	37:41	Loaded from the MSR.
	42:47	Set to zeros (may be used later to distinguish various causes of exception).
	48:63	Loaded from the MSR.

4.5.18 VPU Assist Exception

This exception occurs when operating in Java mode and the input operands or the result of an operation are denormalized.

When this exception is taken, execution resumes at offset x'0000_0000_0000_1700'.

The register settings for this exception are shown in *Table 4-10*.

Table 4-10. Register Settings for VPU Assist Exception

Register	Bits	Setting
SRR0	0:63	Set to the effective address of the instruction that caused the exception.
SRR1	0:32	Loaded from the MSR.
	33:36	Set to zeros.
	37:41	Loaded from the MSR.
	42:47	Set to zeros.
	48:63	Loaded from the MSR.

5. Memory Management

This chapter describes the 970MP implementation of the memory management unit (MMU) specifications provided by the operating environment architecture (OEA) for PowerPC processors. The primary function of the MMU in a PowerPC processor is the translation of logical (effective) addresses to physical addresses (referred to as real addresses in the architecture specification) for memory accesses and I/O accesses (I/O accesses are assumed to be memory-mapped). In addition, the MMU provides access protection on a segment or page basis. This chapter describes the specific hardware used to implement the MMU model of the OEA in each of the 970MP processing units. See the *PowerPC Operating Environment Architecture (Book III)* for a conceptual overview of the memory management model.

Two general types of memory accesses generated by PowerPC processors require address translation—instruction accesses and data accesses that are generated by load-and-store instructions. Generally, the address translation mechanism is defined in terms of the segment descriptors and page tables that the PowerPC processors use to locate the effective-to-physical address mapping for memory accesses. The segment information translates the effective address to an interim virtual address, and the page table information translates the interim virtual address to a physical address.

The segment descriptors, used to generate the interim virtual addresses, reside as segment table entries (STEs) in memory. Each 970MP processing unit uses a segment lookaside buffer (SLB) on-chip that caches recently used segment table entries. In addition, a translation lookaside buffer (TLB) is implemented on each 970MP processing unit to keep recently-used page address translations on-chip.

The MMU, together with the exception processing mechanism, provides the necessary support for the operating system to implement a paged virtual memory environment and to enforce protection of designated memory areas. Exception processing is described in *Chapter 4 Exceptions*. Specifically, *Section 4.3 Exception Processing* on page 133 describes the Machine State Register (MSR), which controls some of the critical functions of the MMUs.

5.1 MMU Overview

The 970MP microprocessor implements the memory management specification of the PowerPC operating environment architecture for 64-bit implementations. The 970MP microprocessor supports a 65-bit virtual address and a 42-bit physical (real) address.

Basic features of the MMU implementation in the 970MP processing unit as defined by the OEA are:

- Support for real addressing mode—Effective-to-physical address translation can be disabled separately for data and instruction accesses.
- Segmented address translation—The 64-bit effective address is translated to a 65-bit virtual address. This 65-bit virtual address space is divided into 4-KB or 16-MB pages, each of which can be mapped to a physical page.

The 970MP microprocessor also provides the following features that are not required by the PowerPC Architecture:

- Unified translation lookaside buffer (TLB)—The 1024-entry, 4-way, set-associative TLB supports:
  - A new large page architecture (16-MB large pages supported).
  - Hardware-based reload (from the L2 cache interface in order to ensure no L1 D-cache impact).
  - Hardware-based update of the reference (R) and change (C) bits in a page table entry (PTE).
  - Parity protection; precise machine-check interrupt on parity error (software fix-up).

  - Recently-used page address translations cached on-chip.
- Segment lookaside buffer (SLB)—The 64-entry, fully associative SLB supports:
  - Software reload of the SLB. An SLB miss results in an interrupt.
  - Loaded by the 32-bit PowerPC Segment Register instructions.
- TLB invalidation—The 970MP microprocessor implements the optional TLB Invalidate Entry (**tlbie**) and TLB Synchronize (**tlbsync**) instructions, which can be used to invalidate TLB entries. For more information about the **tlbie** and **tlbsync** instructions, see *Section 5.3.2.4 TLB Invalidate Entry Instructions* on page 152.
- Little-endian mode is not supported.

Table 5-1 summarizes the MMU features of the 970MP microprocessor, including those defined by the PowerPC Architecture (OEA) for 64-bit processors and those specific to the 970MP microprocessor.

Table 5-1. MMU Feature Summary

Feature Category	Architecturally Defined/ 970MP-Specific	Feature
Address ranges	Architecturally defined	2^{64} bytes of effective address
	970MP-specific	2^{65} bytes of virtual address
		2^{42} bytes of physical address
Page size	Architecturally defined	4 KB
	970MP-specific	16 MB
Segment size	Architecturally defined	256 MB
Memory protection	Architecturally defined	Segments selectable as no-execute
		Pages selectable as user or supervisor and read-only or guarded
Page history	Architecturally defined	Referenced and changed bits defined and maintained
Page address translation	Architecturally defined	Translations stored as PTEs in hashed page tables in memory
		Page table size determined by a mask in SDR1
TLB	Architecturally defined	Instructions for maintaining TLBs (**tlbie** and **tlbsync** instructions in the 970MP microprocessor)
	970MP-specific	1024-entry, 4-way, set-associative TLB (combined for both instruction and data).
Page table search support	970MP-specific	The 970MP microprocessor performs the table search operation in hardware.
Segment descriptors	Architecturally defined	Stored as STEs in hashed segment tables in memory
	970MP-specific	64-entry fully associative SLB
Segment table search support	970MP-specific	The 970MP microprocessor provides support for software reload of the SLB.

5.1.1 Speculative Storage Accesses

The 970MP processing unit is capable of speculatively executing load instructions to non-guarded, cacheable storage. This can occur when a load instruction is encountered on a predicted branch path, or when a logically preceding instruction causes an interrupt. As a result, it is possible for a speculative load that misses in the on-chip cache hierarchy to initiate an external storage request, even if that load instruction is not actually executed as part of the true instruction stream.

5.1.2 Storage Protection

When address translation is enabled, the protection mechanism is controlled by the following bits:

- MSR[PR], which distinguishes between supervisor (privileged) state and user (problem) state
- K_S and K_P, which are the supervisor (privileged) state and user (problem) state storage key bits in the SLB entry, used to translate the effective address
- For instruction fetches only:
  - the N (no-execute) value used for the access
  - the G (guarded) bit in the page table entry used to translate the effective address.

  Thus, for an instruction fetch, access is not permitted if the N value is '1' or if G equals '1'.

5.1.3 Storage Access Modes

Storage access modes are controlled by the write-through/caching-inhibited/memory-coherency enforced/guarded bits (WIMG) bits. The 970MP microprocessor does not support the optional W bit or the optional M bit. All accesses are treated as though W equals '0' and M equals '1' independent of the value of these bits in the page table. Furthermore, when the hardware is performing a change bit update, it will write the W bit as '0' and the M bit as '1'.

Table 5-2 summarizes the treatment of the WIMG bits in the 970MP processing unit:

Table 5-2. Treatment of WIMG Bits in the 970MP Microprocessor

WIMG	Description
x1xx	Treated as WIMG equals '0111', for loads
	Treated as WIMG equals '011x', for stores
x0x1	Treated as WIMG equals '0011'
x0x0	Treated as WIMG equals '0010'

5.1.4 Support for 32-Bit Operating Systems

The 970MP microprocessor supports most of the optional bridge facilities and instructions for 64-bit implementations.

The bridge facility can be used to ease the transition to the PowerPC AS software-managed segment lookaside buffer (SLB) architecture, from either the Segment Register architecture provided by the 32-bit PowerPC implementation or the hardware-accessed segment table architecture provided by the 64-bit PowerPC implementations. The bridge facility permits the operating system to continue to use the 32-bit PowerPC implementation's Segment Register manipulation instructions and to continue to use the Address Space Register (ASR).

Associated with this support, the following optional instructions are supported:

- **mtsr** - Move to Segment Register
- **mtsrin** - Move to Segment Register Indirect
- **mfsr** - Move from Segment Register
- **mfsrin** - Move from Segment Register Indirect
- **mtmsr** - Move to Machine State Register (32-bit)

These instructions allow software to associate effective segments 0 through 15 with any of the virtual segments 0 through 2^{37}-1. SLB entries 0 - 15 serve as virtual Segment Registers, with SLB entry *i* used to emulate Segment Register *i*. The **mtsr** and **mtsrin** instructions move 32 bits from a selected general purpose register (GPR) to a selected SLB entry. The **mfsr** and **mfsrin** instructions move 32 bits from a selected SLB entry to a selected GPR.

5.2 Real Addressing Mode

If address translation is disabled (MSR[IR] equals '0' or MSR[DR] equals '0') for a particular access, the effective address is treated as the physical address and is passed directly to the memory subsystem. These MSR bits are forced to '1' when running in user mode.

The WIMG bits for storage access in real addressing mode are determined as follows. The W and M bits are not supported in the 970MP microprocessor, and are considered to always have values of W equals '0' and M equals '1'. The G bit is always asserted in real addressing mode. For data accesses, bit 23 of Hardware Implementation-Dependent Register 4 (HID4[23]) determines the value of the I bit in real addressing mode. For instruction accesses, HID1[10] can be used to force the value of the I bit to '1', although this value applies to address translation mode as well as to real addressing mode.

5.3 Memory Segment Model

The translation mechanism for segment and page addresses proceeds in the following two steps:

1. Effective addresses are translated to virtual addresses using the segment table.
2. Virtual addresses are translated to real addresses using the page table.

This section highlights those areas of the memory segment model defined by the OEA that are specific to the 970MP microprocessor.

The 970MP microprocessor supports a 65-bit virtual address and a 42-bit physical (real) address.

5.3.1 Segment Table

5.3.1.1 Segment Lookaside Buffer

Each 970MP processing unit contains a unified (combining both instruction and data), 64-entry, fully associative SLB. Information derived from the SLB can also be cached in the instruction effective-to-real-address translation (I-ERAT) cache, or the data ERAT (D-ERAT) cache, or both, along with information from the TLB. As a result, many of the SLB management instructions affect the ERAT caches, as well as the SLB itself (see *Section 5.3.3.1 Instruction Effective-to-Real-Address Translation Cache* and *Section 5.3.3.2 Data Effective-to-Real-Address Translation Cache* on page 154).

Because software (the operating system) manages the SLB, it is possible that multiple entries may be incorrectly set up to provide translations for the same effective address. If an effective address is translated by more than one SLB entry (that is, the effective segment ID [ESID] fields of the entries are identical), a machine check interrupt results with an indication that a parity error occurred when the SLB was accessed. (When this happens, the hardware logically ORs the data in the conflicting entries.) The machine check handler can look at the SLB contents to try to determine if conflicting entries have been provided. When a parity error occurs that is not due to multiple entries, the entire SLB must be reloaded because the Data

Address Register (DAR) does not contain an address indicating which entry caused the parity error. If the source of the error was due to multiple entries, the conflicting entries must be corrected in order for the translation to proceed, which can also be accomplished by reloading the entire SLB with good entries.

5.3.1.2 Address Space Register

The Address Space Register (ASR) is supported in the 970MP microprocessor. Due to the software reload of the SLBs on the 970MP processing unit, this register does not actually participate in any other specific hardware functions on the chip. It has been included as a convenience (and performance enhancement) for the SLB reload software.

5.3.2 Page Table

The 970MP microprocessor supports a maximum page table size of 2^{24} (16 M) page-table-entry groups (PTEG). Because a PTEG is 128 bytes, the page table is limited to 2^{31} bytes (2 GB).

5.3.2.1 Translation Lookaside Buffer

Each 970MP processing unit contains a unified (combined for both instruction and data), 1024-entry, 4-way, set-associative TLB that uses an LRU-based replacement algorithm. In addition, each 970MP processing unit contains a 128-entry, 2-way, set-associative instruction ERAT (single-level effective-to-real-address translation) and a 128-entry, 2-way, set-associative data ERAT. The TLB is a cache of recently used page table entries, and the ERATs are caches that contain translations derived from information in the page table and the SLB. Hardware loads and manages the TLB and ERATs.

For 4-KB pages, the TLB is always indexed with a hashed address (this includes reads from the TLB, writes to the TLB, **tlbie** instructions, and snooped **tlbie** transactions). For 16-MB pages, the TLB is indexed directly with a combination of virtual segment ID (VSID) bits and effective-address (EA) bits. More specifically, the TLB is indexed with the address bits that result from the following:

```
TLB_index(0:7) = (page_size = 4 KB & (EA(36:39) xor EA(44:47)) || EA(48:51)) |
                 (page_size = 16 MB & (VSID(48:51) || EA(36:39)));
```

The 970MP microprocessor supports the hardware update of the storage access recording bits (reference and change) into the memory-based page table.

The execution of **tlbie** instructions, or the detection of **snooped-tlbie** operations off the bus, or both will cause an index-based invalidate to occur in the TLB (that is, all four indexed sets will be invalidated).

Upon power-on, the 970MP processing unit initializes each TLB entry to the invalid state.

5.3.2.2 Large Page Support

In addition to the normal 4-KB page size, the 970MP microprocessor also provides support for 16-MB pages. Translation information for both the 4-KB pages and the 16-MB pages is kept in the TLB.

To avoid accidental large or small page translation aliasing, the 970MP microprocessor implements a bit in HID4 (HID4[61]) to disable the large page facility. It does not permit cache-inhibited accesses to an address in a large page. A PTE that would otherwise match for translation will be ignored for a large page if its WIMG bits indicate that it is cache inhibited.

5.3.2.3 Reference and Change Bits

The 970MP processing unit hardware performs reference and change bit updates to the page table.

The optional W and M bits in the PTE are ignored (assumed to be '01' respectively). If the change bit is updated, the W and M bit in the PTE are set to '01' respectively.

The 970MP microprocessor provides a mode bit (HID4[27]) for determining whether speculative load instructions should reload the TLB in the event of a miss. If this mode bit is set to allow this behavior, the reference bit can be set on behalf of the speculative loads (that is, loads that never actually complete from the perspective of the program). Under no circumstances will the 970MP processing unit speculatively set the page table change bit.

In the 970MP processing unit, load instructions that have a TLB miss still cause the reference bit for the subject page to be set if they are denied access by storage protection or cause one of the following exceptions:

- Address Compare Control Register (ACCR) data storage interrupt (DSI) exception
- Data Address Breakpoint Register (DABR) DSI exception
- I equals '1' DSI exception
- An alignment exception

Similarly, the change bit is set for store types of instructions that cause an ACCR DSI, a DABR DSI, an I equals '1' DSI, or an alignment exception. However, the change bit will *not* be set if a store instruction is denied access by page protection.

5.3.2.4 TLB Invalidate Entry Instructions

The 970MP microprocessor implements the optional **tlbie** instruction described in the architecture (including support for a 16-MB large page).

The **tlbie** instruction, with L set to '0', performs an index-based (congruence class) invalidate of all four sets of the TLB, both sets of the I-ERAT, and both sets of the D-ERAT. The entries in these caches are invalidated without regard for their contents. The **tlbie** instruction, with L set to '1', acts in a similar manner except that it causes a full invalidate of either the I-ERAT, the D-ERAT, or both, if they contain any entries that correspond to large pages.

The **tlbsync** instruction is used to synchronize the completion of the **tlbie** instruction. Only one **tlbsync** instruction is required to synchronize the completion of a group of **tlbie** instructions.

In addition, the **tlbie** instruction broadcasts a tlbie transaction onto the system bus. This transaction includes the full effective address. Other processors or I/O devices that maintain translation caches detect this transaction so that they can invalidate their translation caches based on this address.

In the 970MP processing unit, when a tlbie transaction is detected on the system bus, it will use the address bits to hash index into the TLB, and use the appropriate bits to index into both ERATs and then invalidate all the entries in that congruence class.

The architecture places restrictions on the execution of **tlbie** or **tlbsync** on a given processor after either instruction has been executed on a different processor. Failure to comply with those restrictions can result in undefined system behavior including the possibility of a deadlock.

The 970MP microprocessor also supports a special processor-local form of **tlbie** for use in managing parity errors associated with the TLB. This instruction only supports the 4-KB page size. The format of this implementation-specific instruction is shown in *Figure 5-1 TLB Invalidate Entry Local X-Form (Processor Local Form)* on page 153.

Figure 5-1. TLB Invalidate Entry Local X-Form (Processor Local Form)

tlbiel RB

31	///	10 11	RB	274	/
0	6	9	16	21	31

This instruction operates exactly like the **tlbie** instruction, except that it is not broadcast onto the system bus (it is local to the processor executing the instruction), and it does not support the optional L bit for large pages. To synchronize the completion of this processor local form of **tlbie**, only a **ptesync** is required (**tlbsync** should not be used). This instruction can be executed on a given processor, even if the sequence of **tlbie - sync - tlbsync - sync** is being concurrently executed on a different processor.

5.3.2.5 TLB Invalidate All Instruction

The TLB Invalidate All (**tlbia**) instruction is not implemented in the 970MP microprocessor, and if detected, causes an illegal instruction type of program interrupt. The effects of the instruction can be emulated by executing a series of 256 **tlbiel** instructions with the effective-address bits (0 - 15) set to zero, effective-address bits (36 - 39) held constant, and effective-address bits (44 - 51) incremented through their full range. A sequence of **tlbiel** instructions consumes 17 cycles for each **tlbiel** instruction in the sequence.

5.3.2.6 TLB Synchronize Instruction

The TLB Synchronize (**tlbsync**) instruction forces any previous **tlbie** instructions to complete before the **tlbsync** is allowed to complete.

In the 970MP microprocessor, the **tlbsync** instruction is not broadcast onto the system bus. To ensure that all previous **tlbie**s from the same processor have completed throughout the system, a Page Table Entry Synchronize (**ptesync**) must be used following the **tlbsync** (the **ptesync** must be configured for broadcast onto the system bus, the default mode).

In the 970MP microprocessor, when a ptesync transaction is detected on the system bus, the transaction will not successfully complete until all previously snooped tlbie transactions from the same processor have completed. Each system bus device is responsible for ensuring this behavior.

Software must ensure that only one processor is executing a **tlbsync** instruction at a time, and that the completion of the **tlbie** is properly synchronized. Failure to meet this requirement can result in undefined system behavior including the possibility of a deadlocked system.

5.3.3 Effective-to-Real-Address Translation Caches

5.3.3.1 Instruction Effective-to-Real-Address Translation Cache

Each 970MP processing unit includes a 128-entry, 2-way, set-associative instruction ERAT (I-ERAT) cache for fast translation of instruction effective addresses into physical (real) addresses. Each entry of the I-ERAT cache contains translation information for a 4-KB block of effective storage (even if this section of storage is translated using a large page translation). All instruction accesses use the I-ERAT regardless of whether translation is enabled.

Because the content of each I-ERAT entry is the result of a page table search based on the contents of an SLB entry, in order to maintain consistency with the primary SLB (and Segment Registers, or both), the following instructions will cause all entries in the I-ERAT to be invalidated:

mtsr or **mtsrin**	Used for Segment Register changes in 32-bit operating systems
slbia	Used by software to manage the segment tables
tlbie (with L set to '1')	Used by software to manage large page support
	Note: This instruction only causes the I-ERAT invalidate if the I-ERAT contains translation of any large pages.

The **slbie** instruction causes a class-sensitive invalidate of the I-ERAT. That is, it will invalidate any entries that have a class-bit match with the class indicated by the **slbie** instruction.

In addition, the execution of **tlbie** (with L set to '0') instructions, or the detection of **snooped-tlbie** operations off the system bus, or both cause an index-based invalidate to occur in the I-ERAT. That is, both sets in the congruence class indexed by bits 46 - 51 of the invalidate address will be invalidated.

Note: Upon power-on, each I-ERAT entry is set to the invalid state.

5.3.3.2 Data Effective-to-Real-Address Translation Cache

Each 970MP processing unit includes a 128-entry, 2-way, set-associative data effective-to-real-address translation (D-ERAT) cache for fast translation of data effective addresses into physical (real) addresses. Each entry of the D-ERAT contains translation information for a 4-KB block of effective storage (even if this section of storage is translated using a large page translation). All data accesses use the D-ERAT regardless of whether translation is enabled.

Because the content of each D-ERAT entry is the result of a page table search based on the contents of an SLB entry, to maintain consistency with the primary SLB (and Segment Registers, or both), the following instructions will cause all entries in the D-ERAT to be invalidated.

mtsr or **mtsrin**	Used for Segment Register changes in 32-bit operating systems
slbia	Used by software to manage the segment tables
tlbie (with L set to '1')	Used by software to manage large page support
	Note: This instruction only causes the D-ERAT invalidate if the D-ERAT contains translation of any large pages.

The **slbie** instruction causes a class-sensitive invalidate of the D-ERAT. It will invalidate any entries that have a class-bit match with the class indicated by the **slbie** instruction.

In addition, the execution of **tlbie** (with L set to '0') instructions, or the detection of snooped-**tlbie** operations off the system bus, or both cause an index-based invalidate to occur in the D-ERAT. That is, both sets in the congruence class indexed by bits 46:51 of the invalidate address will be invalidated.

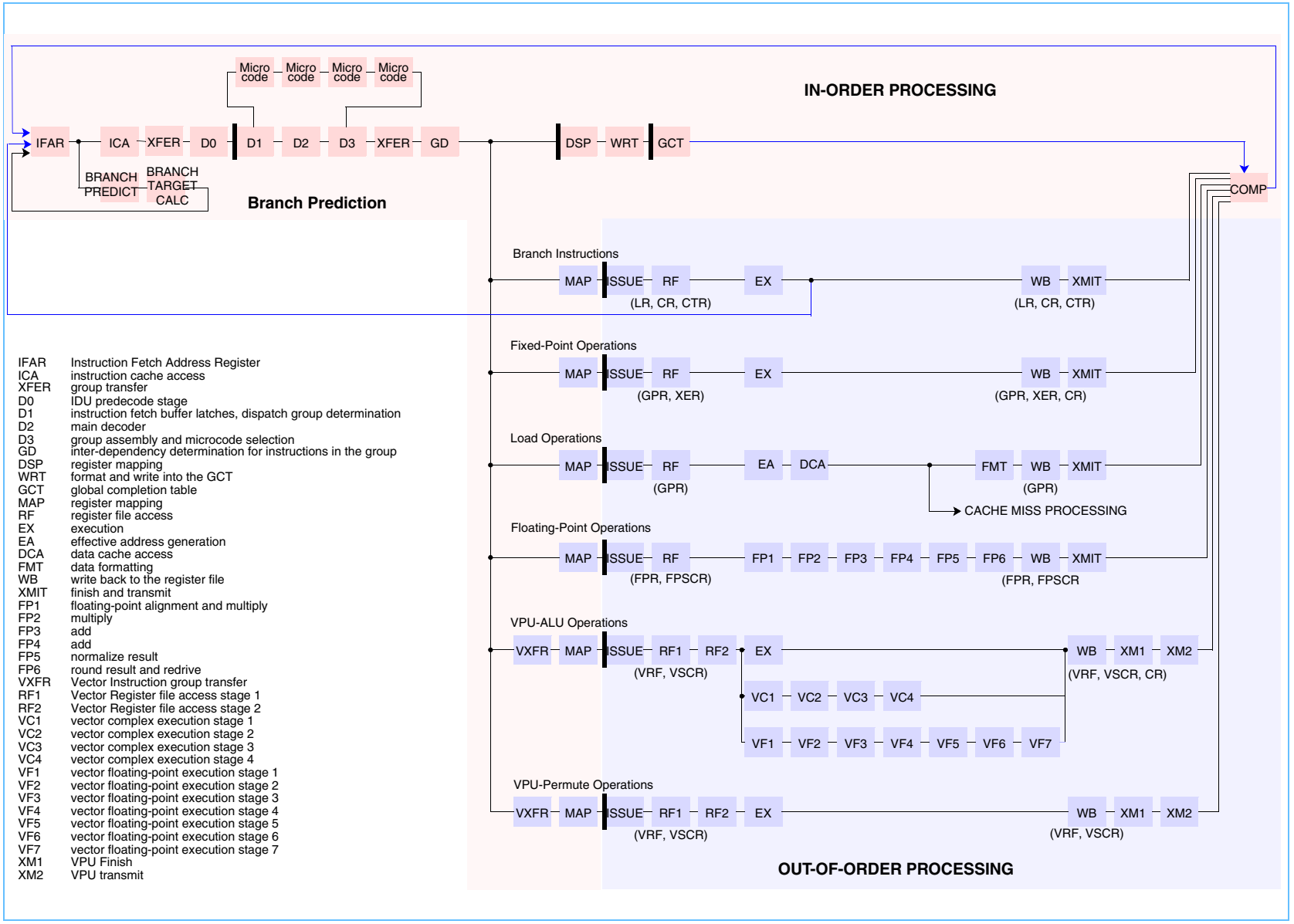
Note: Upon power-on, each D-ERAT entry is set to the invalid state.

6. Performance

6.1 Pipeline Structure of the Microprocessor Core

The pipeline structure for the 970MP processing unit can be subdivided into a master pipeline and several different execution unit pipelines. The master pipeline presents speculative in-order instructions to the mapping, sequencing, and dispatch functions, and ensures an orderly completion of the real execution path. (It throws away any other potential speculative results associated with mispredicted paths.) The execution-unit pipelines allow out-of-order issuing of both speculative and non-speculative operations. The execution-unit pipelines progress independently from the master pipeline and from one another. *Figure 6-1* on page 158 illustrates these pipelines.

Figure 6-1. Microprocessor Core Pipeline Structure



IFAR	Instruction Fetch Address Register
ICA	instruction cache access
XFER	group transfer
D0	IDU predecode stage
D1	instruction fetch buffer latches, dispatch group determination
D2	main decoder
D3	group assembly and microcode selection
GD	inter-dependency determination for instructions in the group
DSP	register mapping
WRT	format and write into the GCT
GCT	global completion table
MAP	register mapping
RF	register file access
EX	execution
EA	effective address generation
DCA	data cache access
FMT	data formatting
WB	write back to the register file
XMIT	finish and transmit
FP1	floating-point alignment and multiply
FP2	multiply
FP3	add
FP4	add
FP5	normalize result
FP6	round result and redrive
VXFR	Vector Instruction group transfer
RF1	Vector Register file access stage 1
RF2	Vector Register file access stage 2
VC1	vector complex execution stage 1
VC2	vector complex execution stage 2
VC3	vector complex execution stage 3
VC4	vector complex execution stage 4
VF1	vector floating-point execution stage 1
VF2	vector floating-point execution stage 2
VF3	vector floating-point execution stage 3
VF4	vector floating-point execution stage 4
VF5	vector floating-point execution stage 5
VF6	vector floating-point execution stage 6
VF7	vector floating-point execution stage 7
XM1	VPU Finish
XM2	VPU transmit

6.2 Branch Processor

6.2.1 Instruction Fetching

In an effort to increase performance, each 970MP processing unit does instruction prefetching before it determines whether particular instructions will actually execute (for more information about the performance aspects of this, see *Section 6.3.1.1 Instruction Prefetching* on page 160). This prefetching follows all of the architectural constraints relative to cache-inhibited and guarded regions of storage. A set of software accessible mode bits are implemented to allow control over the various types of prefetch supported (for more information, see *Section 2.1.2.2 HID Registers (HID0, HID1, HID4, and HID5)* on page 56).

6.2.2 Branch Prediction

The 970MP microprocessor uses several dynamic branch-prediction mechanisms to improve performance. A set of three branch history tables (local, global, and selector) are used to predict the direction of branch instructions early in the pipeline. To improve the efficiency of these predictors, the 970MP microprocessor uses the architected branch-options (BO) field hint bits associated with many of the branch instructions (the so-called "a" and "t" bits).

In addition, for Branch Conditional to Link Register (**bclr**) instructions, a link stack (or call-return stack) is used to predict the target address of the branch. Similarly, for Branch Conditional to Count Register (**bcctr**) instructions, a count cache is used to predict the target address for this type of branch. To improve the efficiency of these address predictors, the 970MP microprocessor uses the architected branch hint (BH) field hints associated with several of the branch instructions. Hardware uses these hints to improve the accuracy of the link stack and the count cache (for more information about these performance enhancement mechanisms, see *Section 6.3.2 Branch Prediction* on page 164).

As the branch instructions progress through the pipeline, eventually they become fully executed. At that point, the hardware determines whether the target address and the direction of the branch matches the actual outcome of the branch. If the prediction was incorrect, the hardware will take the appropriate actions to flush unwanted instructions and results, redirect the pipeline, and update the branch prediction information in the branch history tables. For more information about the performance aspects of branch prediction, see *Section 6.3.2.1 Branch Direction Prediction Using the Branch History Tables* on page 165 and the sections that follow it.

Although the overall performance of the machine is strongly dependent on these branch prediction mechanisms, a set of software accessible mode bits are available to disable these features (see *Section 2.1.2.2 HID Registers (HID0, HID1, HID4, and HID5)* on page 56 for more information).

6.2.3 Special Branch and Link Instruction: BCL 20, 31, $+4

The Branch to Next Instruction with Link Register Update (BCL 20, 31, $+4) instruction specifies a "branch always" to the next instruction, while placing the address of the next instruction in the link register. It can be used to capture the current instruction address for use by software. The 970MP microprocessor supports this usage by not modifying the link stack when this specific form of the branch and link instruction is used. The BCL 20, 31, $+4 instruction does not modify the link stack.

6.2.4 Instruction Cache Touch Hint

The 970MP microprocessor does *not* support the Instruction Cache Touch variant of the branch conditional instructions (this was an implementation-dependent feature of some previous PowerPC AS processor designs). As an alternative, it is possible to use specific sequences of existing instructions to coerce a 970MP processing unit to prefetch cache lines into its prefetch buffers. For more information about these techniques and other coding guidelines, see *Section 6.3.2.2 Branch Prediction Using Static Prediction and the "a" and "t" Bits* on page 166.

6.2.5 Out-of-Order Execution and Instruction Flushes

Each 970MP processing unit uses out-of-order instruction execution. Instructions can be speculative on a predicted branch direction, or speculative beyond an instruction that might cause an interrupt condition. In the event of a misprediction or an interrupt, instructions from the mispredicted path and the results produced by those instructions are discarded, presenting the effect of sequentially executed instructions down the appropriate branch paths and precise exceptions as required.

6.3 Performance Features, Mechanisms, and Hazards

6.3.1 L1 Instruction Cache and Prefetching

6.3.1.1 Instruction Prefetching

To efficiently manage the long processor pipeline and the storage access latency, each 970MP processing unit performs several degrees of instruction prefetching before a particular instruction is due for execution. The simplest and most standard form of instruction fetching works to keep the pipeline as full as possible. Instructions are fetched from the instruction cache (I-cache) (see *Section 6.3.2 Branch Prediction* on page 164, for a discussion on the selection of the specific fetch address) and fed into the pipeline where they work their way through for potential execution and completion. In many cases, the instruction stream is redirected before the fetched instructions make it all the way down the pipeline to completion and, in these cases, the instructions (and speculative results) are cleared from the pipeline.

If a particular instruction fetch misses in the I-cache, several different scenarios are possible. First, the 970MP processing unit looks to see if the instructions are in its instruction prefetch queue and, if so, it steers these instructions into the pipeline as though they came from the I-cache. It also writes the critical sector into the I-cache. If the instructions are forwarded from the prefetch queue, there is a penalty of three cycles when no instructions are fetched. Second, if the instructions were not found in the instruction prefetch queue, a demand fetch reload request (possibly speculative) is sent to the L2 cache subsystem. The L2 cache processes this reload request with high priority and forwards it onto subsequent levels of cache, memory, or both if it misses in these caches.

In addition to these demand-oriented instruction fetching mechanisms, the 970MP processing unit also works to automatically prefetch instruction cache lines that might be referenced soon into its instruction prefetch queue. This queue, which can contain four entries of 128 bytes each, has logic that monitors the demand instruction fetches that miss in the L1 I-cache. If there is an I-cache miss, then it produces prefetches for the next one (if there is a hit in the prefetch queue) or two (if there is a miss in the prefetch queue) sequential cache lines, if the lines are not already in I-cache. Alternately, the prefetch mode can be selected (by properly setting bits in the HID1 Register) to prefetch only the next sequential cache line on an I-cache miss. As these requests return cache lines, they are stored in the instruction prefetch queue so that they do not pollute the

demand-oriented I-cache. The prefetch queue also marks a valid entry in it as "demand" if any sector (see *Section 6.3.1.3 Instruction Fetch Alignment* on page 162) in it has been used for a demand fetch. In cycles when the 970MP processing unit does not read from the I-cache (for example, when there is a cache miss and a prefetch hit, an effective-to-real-address translation (ERAT) miss, a pipeline hold from successive pipeline stages, instruction fetch unit waiting for a I-cache miss to be serviced or a write into a branch history table [BHT] array), a sector from one of these demand entries (if it exists) is written into the I-cache.

Finally, the 970MP microprocessor also supports a special form of the Branch Conditional (**bc), bclr**, and **bcctr** instructions that can be used to act like a software-directed instruction cache touch. A special encoding that looks like a "branch if false" or "branch if true" to previous generation processors, indicates to the 970MP processing unit how the branch direction should be predicted. Use of this along with a condition that is known to be opposite of the prediction sends the instruction fetching along a mispredicted path. By the time the branch instruction is executed and instruction fetching is redirected to the correct path of execution, the (prefetch) request for the cache lines along the mispredicted path will be sent out down the memory hierarchy. See *Section 6.3.2.2 Branch Prediction Using Static Prediction and the "a" and "t" Bits* on page 166 for more details.

6.3.1.2 Predecode Bits

To help the dispatch group formation in the instruction decode unit, instructions received from the L2 interface are partially decoded before being stored into the I-cache or the prefetch buffer. One additional pipeline stage is added between the L2 interface and the I-cache (or the prefetch buffer), which performs this partial decoding and generates 5 predecode bits per instruction (so instructions stored in the I-cache or the prefetch buffer are each 37 bits wide). The predecode bits indicate how instructions should be grouped for dispatching and which instructions should cause an exception. These predecode bits are used by the instruction fetch unit to determine the branches in a fetch group and by the instruction decode unit for dispatch group formation. *Table 6-1* defines the meanings of the 4 predecode bits. The fifth predecode bit is used for instruction match content-addressable memory (IMC) array facility.

Table 6-1. Predecode Bits (Page 1 of 2)

Branch	Split	First	Last	Meaning
0	0	0	0	Unrestricted grouping
0	0	0	1	Last of a group
0	0	1	0	First of a group
0	0	1	1	Only one in group
0	1	0	0	Split and unrestricted grouping
0	1	0	1	Split and last of a group
0	1	1	0	Split and first of a group
0	1	1	1	Split and only one in group
1	0	0	0	Conditional branch
1	0	0	1	Unconditional branch
1	0	1	0	Illegal instruction
1	0	1	1	Floating-Point Machine State Register (MSR) exception
1	1	0	0	Microcoded instruction

Table 6-1. Predecode Bits (Page 2 of 2)

Branch	Split	First	Last	Meaning
1	1	0	1	Reserved
1	1	1	0	Serialize (triggered by IMC)
1	1	1	1	Privileged exception

6.3.1.3 Instruction Fetch Alignment

To simplify the hardware design, instruction fetching occurs on 8-word aligned boundaries. The eight words form an I-cache sector. As instructions are fetched, they are pipelined into an instruction fetch buffer (also known as a cache line buffer), which can hold up to 32 instructions, and which acts as a "shock absorber" between the dispatch logic (below the buffer) and the instruction fetch logic (above the buffer). As instructions are considered for dispatch, the grouping logic can take instructions from two separate fetch groups and put them into a single dispatch group (see *Section 6.3.3 Instruction Decode, Cracking, and Microcode* on page 170, for more information about grouping). It is important to note that, in some performance-oriented short loops, it is possible to see the effect of the 8-word aligned fetching structure. In these cases, it may be important to lay out key instruction sequences to avoid crossing these boundaries.

6.3.1.4 L1 I-Cache Indexing and Replacement Algorithm

The L1 I-cache is direct-mapped, so the replacement algorithm is based solely on the effective address index. On instruction fetches, effective-address bits are used to index into the I-cache, the directory, and the ERAT table. The ERAT contains both the effective addresses and the real addresses. For an ERAT hit, the effective address of the instruction must match the effective address contained in the ERAT entry being indexed, and the ERAT entry must be valid. In addition, the instruction address translation (IR) bit from the MSR at the time of the ERAT miss is stored in the ERAT when the ERAT is loaded on an ERAT miss. This bit must match the corresponding bit in the MSR at the time of instruction fetch for an ERAT hit. The directory contains real addresses, which are compared against the real address coming from the ERAT. The floating-point available (FP) and problem state (PR) (user mode) bits from the MSR at the time of the demand miss for a cache line is also copied into the directory. These three bits are also compared against the corresponding MSR bits during instruction fetch. A cache hit occurs when the real addresses match, the MSR bits match, there is an ERAT hit, the directory entry is valid, and there is no parity error. When these addresses do not match, a cache miss has occurred (see *Load Processing in the CIU and L2* on page 191 for cache miss processing), and the line indexed by the effective-address bits is the one subject to deallocation from the I-cache. It is also important to note that if multiple effective addresses reference the same real address (aliasing), it is possible that these instruction cache lines reside in the I-cache in more than one location. Because the I-cache is a read-only cache (modifications to the instruction stream require special synchronization as defined in the architecture books), no particular hazards exist as a result. However, the effectiveness of the cache may be somewhat diminished if a great deal of localized aliasing is used. In addition, this phenomenon affects how the Instruction Cache Block Invalidate (**icbi**) instruction must operate with respect to the effective-addressed I-cache.

The I-cache is physically a single-ported array capable of either a read or a write in each cycle. On a cache miss, instructions are returned on the L2 cache interface in four 32-byte beats (typically, two beats on successive cycles, followed by several unrelated cycles, followed by the final two beats on successive cycles). The L2 always returns the "critical sector" in the first two beats, and these demand-oriented instructions bypass the I-cache so as to enter the pipeline as quickly as possible. In addition, the cache line is written into one entry of the instruction prefetch queue so that the I-cache itself is available for successive instruction fetches.

The instructions are written to the I-cache during the cycles when the instruction fetching is not using the I-cache (for example, when the pipeline has backed up or when another I-cache miss occurs). In this way, the writes to the I-cache are hidden and do not interfere with normal instruction fetches.

6.3.1.5 ICBI Instruction Handling

The I-cache is indexed with effective-address bits, and tagged with real address bits. The **icbi** instruction requires that I-cache entries associated with the real address produced by the **icbi** instruction are removed from the cache. As a result, the **icbi** instruction must look in all possible locations where a particular real address may reside (recall that the I-cache is indexed with effective-address bits and that aliasing may cause a particular real address cache line to reside in multiple spots within the I-cache). Because the I-cache is 64 KB in size, and page granularity is on 4-KB blocks, the **icbi** instruction has 16 possible effective-address indexed locations that must be checked. To avoid blocking the use of the I-cache during these required lookups for the **icbi** instruction, each 970MP processing unit has two copies of the single-ported I-cache directory. In addition, only entries that match the real address will actually be invalidated. This allows the **icbi** instruction (either locally initiated or snooped off the system bus) to block the use of the I-cache only for *n* cycles, where *n* is the number of effective-address indexed locations that has the real address produced by the **icbi**.

After the storage subsystem (STS) sends an ICBI operation to the instruction fetch unit (IFU), a subsequent ICBI operation will not be issued until 32 cycles later (see *ICBI* on page 107).

The effective address of an **icbi** instruction is computed and translated by the load/store unit (LSU), and the real address is sent to the STS to be broadcast on the bus interface unit (BIU). The LSU also increments a counter indicating the number of locally generated **icbi** instructions that are yet to be completed by the IFU. The STS informs the IFU to process an **icbi** (and supplies the corresponding real address) and indicates whether the **icbi** is local. After the IFU completes the **icbi**, it indicates to the LSU the completion of a local **icbi** and to the STS the completion of any **icbi**. If a locally generated **icbi** is completed by the IFU, the LSU decrements its counter. When the counter is zero and there is no **icbi** in the store reorder queue (SRQ) yet to be completed, all of the locally generated **icbi** instructions are guaranteed to have been completed for context synchronization.

6.3.1.6 Instruction Fetch Address Translation

As previously mentioned, the effective-address bits are used to index into the I-cache, the directory, and the ERAT table. In the event of an ERAT miss, if the instruction is being fetched speculatively (that is, there is at least one branch instruction that has been fetched, but not committed or cleared from the pipeline), then the instruction fetch logic sends a speculative I-ERAT reload request to the second-level memory management facility, which contains the unified translation lookaside buffer (TLB) and segment lookaside buffer (SLB). If both a TLB hit and an SLB hit occur, the appropriate information required to reload the ERAT is returned, the ERAT is reloaded, and the instruction fetch is attempted again. If either a TLB or an SLB miss occurs and a speculative TLB walk is allowed (a feature that can be turned off through the proper setting of the HID1 Register), the hardware-based page table reload state machine engages and works to reload the TLB. If a speculative TLB walk is turned off, the request is denied and the instruction fetching logic must wait until the requested instruction becomes non-speculative.

Alternatively, if the instruction being fetched is non-speculative (that is, all the branch instructions that have been fetched are either committed or cleared), then the instruction fetching logic sends a non-speculative I-ERAT reload request to the second-level memory management facility. If a TLB and an SLB hit occur, the information is returned as in the speculative case. However, in this scenario if a TLB miss occurs, the hardware-based reload state machine of the page table engages immediately and works to reload the TLB. Once

reloaded, the second-level memory management unit returns the information required to reload the ERAT to the instruction fetch logic. In the event of an SLB miss (the 970MP processing unit requires a software-based reload of the SLB) or a TLB fault of any kind (such as a page fault or protection violation), an indicator is sent back to the instruction fetcher, which then introduces a special interrupt-causing instruction into the pipeline. When this special instruction gets to the completion stage of the pipeline, the appropriate interrupt is taken.

The I-ERAT is a 128-entry, 2-way set-associative translation cache, which uses a first-in-first-out (FIFO) replacement algorithm. In this algorithm, one bit is kept per congruence class and is used to indicate which of the two entries was loaded first. As the name "replacement algorithm" implies, the first entry loaded is the first entry targeted for deallocation when a new entry needs to be loaded into that congruence class.

Each entry in the ERAT provides translation for a 4-KB block of storage. If a particular section of storage is actually mapped by a large-page TLB entry, each referenced 4-KB block of that large page will occupy an entry in the I-ERAT (that is, a large-page translation is not directly supported in the ERAT). Although there is no translation required for fetching instructions in MSR[IR] equals '0' mode, an entry in the ERAT is created nevertheless for logic simplification. No ERAT entry is created for the pages for which the page table entry is marked guarded (G equals '1'). Invalidation of a large page TLB entry is achieved by invalidating the entire I-ERAT, whereas invalidation of an SLB entry is achieved by invalidating all the I-ERAT entries that match the class bit for the SLB entry; both of these invalidation operations take one cycle.

6.3.1.7 Instruction Fetching while in Real Mode (MSR[IR] equals '0')

Instruction fetches can either be prefetches or demand fetches. A prefetch is typically initiated well ahead of the executing instruction stream and is intended to move instructions from potential future paths up in the cache hierarchy. The instruction fetching machine is designed so that the prefetches can either return data or not return data. A demand fetch is initiated when the instruction fetch machine has decided which path it is pursuing, and that instruction (or block of instructions) is the next one required to keep the pipeline fully progressing. Note that this type of fetch can still be speculative, and that from an instruction completion point of view, there are still many instructions ahead of it in the pipeline. Demand fetches that turn out to be not needed (that is, a branch is resolved in a way that redirects the instruction fetcher down an alternate path) can be dynamically turned into simple prefetches if the redirection is known soon enough.

The 970MP processing unit does not make any distinction between speculative and non-speculative instruction fetches and sends instruction fetch requests to the L2 whenever there is a translation in the I-ERAT. Because speculative instruction fetching from guarded storage can cause potential problems, it is handled by placing proper restrictions on what entries can be loaded into the I-ERAT. As mentioned earlier, there is no I-ERAT entry for pages marked as guarded. An attempt to fetch instructions from a storage section with G set to '1' causes an I-ERAT miss and a subsequent instruction storage interrupt (ISI) exception. Instruction fetching with IR set to '0' is allowed to proceed speculatively, if there is an I-ERAT hit. However, on an I-ERAT miss with IR set to '0', translation is not initiated speculatively (that is, until it is known that all branch instructions that have been fetched by the core have either been committed or cleared).

6.3.2 Branch Prediction

In each cycle, up to eight instructions are fetched from the instruction cache. From there, these instructions are sent to the 4-stage instruction decode logic (see *Section 6.3.3 Instruction Decode, Cracking, and Microcode* on page 170). Then they are sent to the branch prediction logic. The branch prediction logic scans all the fetched instructions, looking for up to two branches per cycle. Depending upon the branch type found, various branch prediction mechanisms engage to help predict the target address of the branch, or the branch direction, or both. More specifically, branch target addresses for **bclr** and **bcctr** instructions can be predicted using the link stack and the count cache mechanism (target addresses for absolute and relative branches are

computed directly as part of the branch scan function). Branch direction prediction (taken or not taken) is done by using a set of three branch history tables. Additional details of these mechanisms are described in the following subsections.

It is important to note that all conditional branches are predicted in the 970MP microprocessor (even if the condition is resolved well ahead of time or the value of the Link or Count Register is known when the branch-to-link/count instruction is fetched) and no unconditional branches are predicted. As branch instructions flow through the rest of the pipeline, and ultimately execute in the branch execution unit, the actual outcome of the branches is determined. At that point, if a prediction was correct, the branch instruction is just completed like all other instructions. If the prediction was incorrect, the instruction fetch logic issues a flush and redirects the pipeline down the corrected path.

6.3.2.1 Branch Direction Prediction Using the Branch History Tables

Each 970MP processing unit uses the following set of three branch history tables to predict the direction of branch instructions:

- Local predictor
- Gobal predictor
- Selector table

The first table, called the local predictor, is similar to the traditional branch history table. It is a 16-K entry array that is indexed by the address of the branch instruction to produce a 1-bit predictor. The 1-bit predictor indicates whether the branch direction should be taken or not-taken. The second table, called the global predictor, works to predict a branch based on the actual path of execution to reach the branch. The path of execution is identified by an 11-bit vector, 1 bit per fetch group (that is, the group of instructions fetched in a cycle), for each of the previous 11 fetch groups. This vector is referred to as the global history vector. Each bit in the global history vector indicates whether the next group of instructions fetched are from a sequential cache sector ('0') or not ('1'). The global history vector captures this information for the actual path of execution through these sectors. If there is a redirection of instruction fetching, some of the fetched group of instructions are discarded and the global history vector is corrected immediately. The global history vector is hashed (by bitwise XOR) with the address of the branch instruction in order to index into the 16-K entry global history table producing another 1-bit branch direction predictor. Similar to the local predictor, this 1-bit global predictor is an alternate indicator of whether the branch should be predicted to be taken or not-taken. Finally, the third table, called the selector table, keeps track of which of the two prediction schemes works better for a given branch and is used to select between the local and the global predictions. The 16-K entry selector table is indexed exactly the same way as the global history table is indexed in order to produce the 1-bit selector. This combination of branch prediction tables has been shown to produce very accurate predictions on a wide range of workload types.

The 970MP processing unit also has a compress history mode (selected by a HID1 bit) to compress the global history vector. This is useful for improving the branch prediction accuracy of workloads that exhibit too many sequential fetch groups (that is, numerically intensive workload with loop unrolling). In the history compression mode, up to four successive zeros in the global history vector are compressed into one zero. This effectively allows the global history vector to capture a larger segment of the path of execution leading to the branch being predicted.

If the first branch encountered in a particular cycle is predicted as not taken, then the 970MP processing unit can predict and act on a second branch in the same cycle. In this case, the machine will register both branches as predicted (for subsequent resolution at branch execution), and it will redirect the instruction fetching based on the second branch.

As branch instructions are executed and resolved, the branch history tables (as well as the other predictors described below) are updated to reflect the latest and most accurate information. Unconditional branches[1] and statically predicted conditional branches[2] do not have an entry in the BHTs. Therefore, they do not cause any BHT update. Because the majority of the branches do not change directions very often, BHT entries are read much more frequently than written. To take advantage of this observation and save precious silicon area, all three BHTs are single ported. A BHT write takes place only when the bit read at the time of making the branch prediction is different from what it should be after the branch instruction is resolved. Instruction fetching is stopped for one cycle when a BHT write takes place. Because immediately after a branch misprediction the instruction pipeline is likely to have very few instructions, the cycle immediately following the branch misprediction is used to fetch instructions from the corrected address. The subsequent cycle is used for updating the BHTs. Because BHTs are not updated immediately after a branch misprediction, for loops that fit in a cache sector, the loop-ending branch may be mispredicted twice in a row in the first two iterations of the loop.

6.3.2.2 Branch Prediction Using Static Prediction and the "a" and "t" Bits

For some conditional branches, the software knows what the branch direction prediction ought to be. The 970MP processing unit allows the software to override the dynamic branch prediction in such cases. Software communicates its wish to override dynamic branch prediction by setting the "a" bit in the BO field. If the "a" bit is '0', then dynamic branch prediction as described above is used. If the "a" bit is '1' for a branch, dynamic branch prediction is not used and no entry is updated in the branch history table when such a branch is executed. The static branch direction prediction itself is communicated by properly setting the "t" bit in the BO field ('1' for taken and '0' for not-taken). Software is expected to use this feature only for those conditional branches for which it believes that software branch prediction will result in at least as good a performance as the hardware branch prediction.

Three separate cases have been identified where the software should override hardware branch prediction for improved performance.

- *When the conditional branch is known by the software to be almost always uni-directional*. For example, branches that guard segments of code that are only executed when a rare event occurs. Another example is a branch that guards the loop body. As explained in *Section 6.3.2.1* on page 165, if the fetch history of a loop (with all its iterations) fits in the 11-bit global history vector, then once the history is known (after the first execution of the loop), the 970MP processing unit is likely to predict such a branch with perfect accuracy for all the successive executions of the loop. However, if the fetch history of the loop (with all its iterations) does not fit in the global history vector, then the software branch prediction, by setting the "a" and the "t" bits, will provide better performance.
- *For the branches that close out a lock acquisition sequence*, it is desirable to force the branch prediction to be *not* taken. This provides the best performance for the most common case where the lock is successfully acquired. Even if the lock is not successfully acquired on this iteration, it is still best to assume (from a branch prediction standpoint) that it will be acquired in the next iteration. Note that, left alone, if the lock is not acquired in the first iteration, the branch history mechanism would work to update the pre-

1. Includes branches with the BO field set to '1z1zz'.
2. Branches with the "a" bit set to '1'; see *Section 6.3.2.2 Branch Prediction Using Static Prediction and the "a" and "t" Bits* on page 166.

diction to predict *taken* (that is, predict lock acquisition failure and cause more Load Word and Reserve Indexed [**lwarx**] traffic) for the next iteration.

```
top:  lwarx
       add
       stwcx
       bc top     <-- 970MP processing unit will predict this branch to be not taken, through
                      software directives that properly set the "a" and "t" bits.
```

- *To force a conditional branch to always be mispredicted in order to initiate instruction prefetching.* This allows some instructions to be speculatively executed or processed to some extent by the instruction fetch logic before they are discarded. The instruction in the (incorrect) predicted path can be used as a hint instruction to the memory subsystem. For example, software prefetching of instructions from location "Line_to_touch" can be initiated by forcing a branch misprediction as follows (the "a" bit in the **bc** instruction indicates must agree with static prediction):

```
Short distance touches:
       bc Line_to_touch          // Static prediction taken, but CR bit is set "not-taken"
Long distance touches:
       bc Next                   // Static prediction not-taken, but CR bit is set "taken"
       b Line_to_touch           // Initiate prefetch for cache line "Line_to_touch"
   Next:  ...                    // Instructions in the actual instruction stream
```

  This type of software prefetching is useful if the line to prefetch is in memory. Due to the high penalty for branch misprediction in the 970MP processing unit, it may not be beneficial if the referenced line is already in the L2 cache. It may even be harmful if the referenced line is already in the I-cache. It is beneficial if the compiler makes special attempts to schedule code around such a branch that reduces the misprediction penalty. Attempts can be made to reduce the forced branch misprediction penalty. For example:

  - Set the Condition Register (CR) bit used by the **bc** as early as possible.
  - Schedule such a branch in a code segment where there are relatively few branches so that the branch does not wait too long in the branch issue queue behind other branches.
  - Try to overlap a likely data cache (D-cache) miss with the forced branch misprediction.
  - Schedule such a branch after an existing long chain of flow dependency.

6.3.2.3 Address Prediction Using the Link Stack

The 970MP processing unit uses a link stack to predict the target address for a branch-to-link instruction that it believes corresponds to a subroutine return. By setting the hint bits in a branch-to-link instruction, software communicates to the processor whether the instruction represents a subroutine return, or a target address that is likely to repeat, or neither (see *Table 6-2* on page 168).

When instruction fetch logic fetches a branch and link instruction that is either unconditional or conditional but is predicted taken, it pushes the address of the next instruction into the stack. When it fetches a branch-to-link instruction with a taken prediction and with hint bits indicating a subroutine return, the address is popped off the stack, and instruction fetching starts from the popped address.

Speculative execution can alter the link stack (both its pointer and its contents). The exact nature of the alteration depends on the sequence of the stack-modifying branch instructions that are cleared from the system on a mis-speculation. For example, if only a branch and link instruction is cleared, then the link stack pointer is one higher than what it should be. However, if the cleared instructions consist of a branch-to-link instruction followed by a branch and link instruction, then the pointer is not altered but the content is.

Because branch-to-link instructions are fairly common and the branch misprediction penalty is high, the 970MP processing unit employs an extensive speculation tolerance mechanism in its link stack implementation, which allows it to recover the link stack under most circumstances. To recover the stack pointer at mis-speculation, the value of the stack pointer at the time a branch is scanned by the instruction fetch logic is stored in the branch information queue along with other information for the branch. To recover the contents of the link stack, the 8-entry link stack is implemented as a 16-entry renamed Link Stack Register file (see *Section 6.3.5 Register Renaming* on page 183 for more information). A pop followed by a push does not overwrite the popped value. Instead the pushed value is written into a different (renamed) physical location in the register file. Thus, the popped value is still available in the register file, and recovered on mis-speculation.

*Table 6-2. Handling **bclr** and **bclrl** Instructions*

Instruction	BH Field	970MP Design	PowerPC Architecture
bclr, **bclrl**	00	If the branch is predicted taken, the link stack is popped and the popped address is used as the predicted target address. No address is pushed in the link stack.	The branch is a subroutine return.
bclr, **bclrl**	01	If the branch is predicted taken, the target address is predicted using the count cache. The count cache is updated when the branch is executed, if the branch is resolved as taken. No address is pushed in the link stack.	Target address is likely to repeat.
bclr, **bclrl**	10	Same as BH equals '00'.	Reserved.
bclr, **bclrl**	11	If the branch is predicted taken, the target address is predicted using the count cache. The count cache is not updated when the branch is executed, and no address is pushed in the link stack.	Target is not predictable.

6.3.2.4 Address Prediction using the Count Cache

The target address of a Branch to Count (**bcctr**[**l**]) instruction is often repetitive and can be predicted if the address is saved in a cache from an earlier execution of the same instruction. This is also true for some of the Branch-to-Link (**bclr**[**l**]) instructions, which are not predictable through the use of the link stack, because they do not correspond to a subroutine return. By setting the hint bits appropriately, software communicates to the hardware whether the target address for such branches is predictable using a cache (see *Table 6-3* on page 169).

The 970MP processing unit uses a 32-entry, tagless, direct-mapped cache, called a count cache, for predicting the target of a **bclr**[**l**] or **bcctr**[**l**] instruction whose target address is likely to repeat. Each entry in the count cache can hold a 62-bit address. When a **bclr**[**l**] or **bcctr**[**l**] instruction is executed, for which the software indicates that the target is predictable using such a cache, the target address is written in the count cache. When such an instruction is fetched, the target address is predicted using the count cache.

The entry in the count cache to write to or to read from is determined by address bits 54 through 58 of the branch instruction. If two **bcctr**[**l**] instructions are in a loop and they map to the same entry in the count cache, they can destroy each other's target address in the count cache. In such cases, it may be better to encode the BH field of the more important **bcctr**[**l**] instruction as '00' and the other as '11'.

*Table 6-3. Handling **bcctr** and **bcctrl** Instructions*

Instruction	BH Field	970MP Design	PowerPC Architecture
bcctr, **bcctrl**	00	If the branch is predicted taken, the target address is predicted using the count cache. Update the count cache when the branch is executed, if the branch is resolved as taken. For the **bcctrl** instruction, if the branch is predicted taken, push the link stack in the address of the next sequential instruction when the **bcctrl** instruction is fetched.	Target address is likely to repeat.
bcctr, **bcctrl**	01	Same as BH equals '00'.	Reserved
bcctr, **bcctrl**	10	If the branch is predicted taken, the target address is predicted using the count cache. For the **bcctrl** instruction, if the branch is predicted as taken, push the link stack in the address of the next sequential instruction when the **bcctrl** instruction is fetched. Note that the count cache is not updated when the branch is executed.	Reserved
bcctr, **bcctrl**	11	Same as BH equals '10'.	Target is not predictable

6.3.2.5 Round-Trip Branch Processing

The instruction fetch logic can fetch up to eight instructions in a given cycle and forward them down the pipeline to the instruction decode logic. The instruction fetch logic scans this group of instructions in the next cycle to determine the position of all the branches in the group and the predictions for them. At the same time, it fetches the next sequential cache sector of eight instructions if there is no I-cache miss, I-ERAT miss, fetch redirection, or pipeline hold (for example, due to successive stages not having enough resources to process the instructions already in progress). The next sequential I-cache sector is fetched with the assumption that there is no unconditional branch, nor a conditional branch that was predicted taken in the previous group of instructions it fetched. If the assumption is correct, then the instruction fetch proceeds unhindered. If the assumption is incorrect, a BHT redirection event occurs, when the instruction fetch logic instructs the instruction decode logic to discard the instructions it sent after that branch and starts fetching from the target of that branch. If it is a branch-to-link/count instruction, then the target of the branch is predicted based on the prediction mechanism described earlier. Otherwise, the target of the branch is calculated from the instruction and its address. On a BHT redirection, two cycles worth of instruction fetching may be lost.

Instruction fetch logic scans the instructions obtained from a cache sector and processes up to two branch instructions per cycle. The second branch is not processed if the first branch is unconditional or predicted taken. To reduce the complexity of the design, if the instruction fetch logic scans more than two conditional branches in the cache sector and the first two branches are both predicted as not-taken, then a BHT redirection event is generated and all instructions fetched after the current sector are discarded and refetched. This may result in a loss of two cycles worth of instruction fetching. During scanning, if a branch and link instruction is found that is unconditional or predicted taken, then the address of the next instruction is pushed into the link stack (see *Section 6.3.2.3 Address Prediction Using the Link Stack* on page 167 for more details). A lot of information about the scanned branch (such as, the direction predicted, the target predicted, its position, global history vector used to predict the branch, and so on) is also collected at this time and stored in a queue, referred to as the branch information queue. This information is used when the branch is executed to determine whether the branch has been predicted correctly or not. If the branch has been incorrectly predicted, the information is used to determine where to start fetching from next, how to update the branch history tables, and how to correct the link stack.

Instruction decode logic also scans through the instructions and forms groups (see *Section 6.3.3 Instruction Decode, Cracking, and Microcode* on page 170 for the constraints in forming the groups). Each group consists of at most five internal operations with at most one branch, which can only be at the fifth slot of the

instruction group. After the instructions are decoded and the groups are formed, each group enters the dispatch stage in program order, where the architected registers used by these instructions are renamed to physical registers (see *Section 6.3.5 Register Renaming* on page 183 for more information). For a branch instruction, the CR Register field and the Link and the Count Register can be renamed. A register file of 32 entries (each entry is 4 bits wide) is used to rename the CR field, and a register file of 16 entries (each entry is 64 bits wide) is used to rename the Link and the Count Registers (see *Section 6.3.6 Instruction Issue and Register File Access* on page 186).

For each group dispatched, an entry is made in the global completion table (GCT). The instructions then proceed to the issue stage. If the dispatched group has a branch in it, it takes up an issue queue slot in the 12-entry branch issue queue. If the branch issue queue is full, no dispatching takes place until an entry becomes available. An instruction waits in the issue queue until it becomes the oldest instruction in the queue for which all the sources are available. So the order of execution of the branch instructions may be different from the program order. When a branch is ready for execution, the branch execution logic reads the entry in the branch information queue for the branch, the CR Register field, and/or the Link Register, and/or the Count Register (depending on the branch instruction) from the CR Register file, and the Link-Count Register file respectively. It executes the branch in the following cycle. The execution may update either the Link Register, Count Register, or both. After execution, the entry in the branch issue queue is freed up.

The primary purpose of a branch instruction is to direct the flow of control through a program. When a branch is fetched, the instruction fetch logic makes every effort to predict the flow of control by predicting the direction or the target of the branch, or both. When the branch is executed, the outcome is compared with the prediction made at the fetch stage (predictions are found in the branch information queue). If the predictions are correct, instruction completion logic waits for the rest of the instructions in the group to finish execution. When they all complete, a tag identifying the group is sent to all the queues for their cleanup. If there is a branch in the group, the corresponding entry in the branch information queue is freed up. Groups complete in program order.

If the branch is mispredicted, all the instructions in the pipeline that are fetched after the branch instruction are cleared from the system. New instructions from the new address, as determined by the executed branch instruction, are fetched. The clearing involves restoring all the queues and the mapper register files to the state that is consistent with not having fetched or executed any instruction after the branch.

Because of its deep pipeline, the branch misprediction penalty is high in the 970MP processing unit. If a mispredicted branch is executed before some of its earlier instructions (in program order), the pipeline will not be completely dry after the misprediction. These earlier instructions can keep the execution units busy while the instruction fetch logic brings new instructions, thus reducing the penalty to some extent. The average branch misprediction penalty depends on many factors including when the associated CR bit becomes available, the number of instructions that survive a misprediction, and associated cache misses. However, once a branch is executed and found to be mispredicted, it takes a minimum of twelve cycles for a new instruction from the redirected path to come down the pipeline and get executed.

6.3.3 Instruction Decode, Cracking, and Microcode

After instructions are fetched from the instruction cache, they are placed in the instruction fetch buffer. The instruction fetch buffer can hold 32 instructions. Up to five instructions per cycle are pulled from the instruction fetch buffer and sent through the 3-stage instruction decode pipeline to form a dispatch group. A dispatch group can have instructions from two consecutive quadwords. The instruction decode pipeline serves to break up some of the more complex instructions into internal operations, and to format (expand) the resulting

internal operations to match the dataflow of the inner core. For instructions that need to be broken into two internal operations, the instruction decode pipeline provides in-line dataflow to perform this instruction cracking.

For instructions that need to be broken into more than two internal operations, the 970MP processing unit uses an extended decode pipeline that makes use of a microcode-based template to emulate the full effects of the instruction with one or more groups of four (non-branch) internal operations. Only one instruction of this type can be pulled from the instruction fetch buffer into the extended decode pipeline per cycle. If the preceding instructions are non-branch and non-microcode instructions, then up to four of these instructions can be pulled from the instruction fetch buffer concurrently.

In addition, for some instructions that encounter difficult situations during execution (that is, certain types of unaligned storage references), the instruction decode pipeline cooperates with the load/store unit and the instruction sequencing unit to flush the first attempt on these instructions, and then redispatch them using the template-based microcode and feedback from the first attempt. This is described more fully in *Section 6.3.3.2 Microcode Template-Based Instruction Cracking* on page 175 and *Section 6.3.3.3 Run Time Feedback-Based Instruction Cracking* on page 178.

Certain conditions introduce a bubble of one or more empty cycles in the flow of dispatch groups from the instruction decode unit. *Table 6-4* summarizes the decoder pipeline bubble conditions.

Table 6-4. Pipeline Bubbles Due to Microcode

First Instruction	Second Instruction	
	Non-Microcode	Microcode
Non-microcode	0	1 or 2
Microcode (one group)	0	0
Microcode (more than one group, fixed length)	0	2
Microcode (variable length)	1	3

A relatively rare scenario causes an additional 1-cycle bubble in all cases. This occurs when a group consumes the last instruction in quadword 0, as well as all valid instructions in quadword 1. The additional bubble occurs because the decoder pipeline control logic can retire at most one quadword from the instruction fetch buffer in each cycle. See *Section 6.3.3.2 Microcode Template-Based Instruction Cracking* on page 175 for details about the conditions that cause bubbles preceding or following microcode instructions.

6.3.3.1 In-line Dataflow Instruction Cracking

Up to five instructions are pulled from the instruction fetch buffer and presented to five unique decode pipes based on the predecode bits associated with the instructions. These predecode bits determine how many instructions are pulled from the instruction fetch buffer and how they are used to form a dispatch group.

The following types of instructions are marked as SPLIT and require two decode pipes:

Instructions that require two destinations of the same type	General Purpose Register (GPR), Floating-Point Register (FPR), Integer Exception Register (XER), Condition Register (CR), Link Register (LR), Count Register (CTR)
Instructions that require more than two sources of the same type	GPR or CR

If the first three decode pipes are already in use by previous instructions in the dispatch group, the dispatch group will be prematurely ended. The instruction in question will form the first two internal instructions of the next dispatch group. The fixed-point divides are a special case. They are cracked into two internal operations, a no-op and a fixed-point divide, so that the divide internal operation is presented to the second dispatch slot.

Instructions that require specific execution units are marked as FIRST. They will end a preceding dispatch group prematurely if that group would not end because of another constraint such as a branch, a microcoded instruction, or because all four non-branch/microcode decode pipes are in use. Instructions that require this behavior include most instructions that update non-renamed resources or Condition Register logicals, or use the fixed-point divider.

Instructions that require that no other instructions follow in the same dispatch group are marked as LAST and will end the present dispatch group prematurely. These instructions include updates to non-renamed resources.

Table 6-5. Cracked Instructions

Instruction
addc
adde.
addeo
addic.
addme.
addmeo
addo.
addze.
addzeo
crand[a]
crandc[a]
creqv[a]
crnand[a]
crnor[a]
cror[a]
crorc[a]
crxor[a]
divd
divdo
divdu
divduo
divw
divwo
divwu
divwuo
dst
dstst
dststt
dstt
extsb.
extsh.
extsw.

Instruction
lbzu
ldu
lfdu
lfdux
lfsu
lfsux
lha
lhax
lhzu
lwa
lwax
lwzu
mulhd.
mulhdu.
mulhw.
mulhwu.
mulld.
mulldo.
mullw.
mullwo.
nego.
rldcl.
rldcr.
rldic.
rldicl.
rldicl.
rldicr.
rldimi.
rlwimi
rlwinm.
rlwnm.
sld.

Instruction
slw.
srad.
sradi.
sraw.
srawi.
srd.
srw.
stbu
stbx
stdu
stdx
stfdu
stfdux
stfsu
stfsux
sthbrx
sthu
sthx
stwbrx
stwu
stwx
subfc
subfe.
subfeo
subfme.
subfmeo
subfo.
subfze.
subfzeo

a. CR logicals are cracked if **crb**D is not equal to **crb**B.

Table 6-6. Instructions with Group Formation Restrictions

Instruction	First	Last
b		last
ba		last
bc		last
bca		last
bcctr		last
bcctrl		last
bcl		last
bcla		last
bclr		last
bclrl		last
bclrl		last
bl		last
bla		last
crand	first	
crand	first	
crandc	first	
crandc	first	
creqv	first	
creqv	first	
crnand	first	
crnand	first	
crnor	first	
crnor	first	
cror	first	
cror	first	
crorc	first	
crorc	first	
crxor	first	
crxor	first	
dcbf	first	last
dcbst	first	last
dcbz	first	last
divd	first	
divdo	first	
divdu	first	
divduo	first	
divw	first	
divwo	first	
divwu	first	
divwuo	first	
dss	first	last
dssall	first	last
dst	first	last
dstst	first	last
dststt	first	last
dstt	first	last
eieio	first	last
fabs.	first	last
fadd.	first	last
fadds.	first	last
fcfid.	first	last
fctid.	first	last
fctidz.	first	last
fctiw.	first	last
fctiwz.	first	last
fdiv.	first	last
fdivs.	first	last
fmadd.	first	last
fmadds.	first	last
fmr.	first	last
fmsub.	first	last
fmsubs.	first	last
fmul.	first	last
fmuls.	first	last
fnabs.	first	last
fneg.	first	last
fnmadd.	first	last
fnmadds.	first	last
fnmsub.	first	last
fnmsubs.	first	last
fres.	first	last
frsp.	first	last
frsqrte.	first	last
fsel.	first	last
fsqrt.	first	last
fsqrts.	first	last
fsub.	first	last
fsubs.	first	last
icbi	first	last
isync	first	last
ldarx	first	last
lwarx	first	last
lwsync	first	last
mcrf	first	
mcrfs	first	last
mfcrf	first	
mffs	first	last
mffs.	first	last
mfmsr	first	
mfspr	first	
mfsr	first	
mfsrin	first	
mftb	first	
mfvrsave	first	
mfvscr	first	last
mtfsb0	first	last
mtfsb0.	first	last
mtfsb1	first	last
mtfsb1.	first	last
mtfsf	first	last
mtfsf.	first	last
mtfsfi	first	last
mtfsfi.	first	last
mtmsr	first	last
mtmsrd	first	last
mtspr	first	last
mtspr (LR, CTR)	first	
mtvrsave	first	last
mtvscr	first	last
ptesync	first	last
rfid	first	last
sc	first	last
slbmfee	first	last
slbmfev	first	last
sync	first	last
tlbsync	first	last

6.3.3.2 Microcode Template-Based Instruction Cracking

The microcode decode pipeline generates up to four internal instructions per cycle, which are used to emulate the original instruction. This pipeline is two cycles longer than the decode pipeline for other instructions. However, one of the two cycles is hidden when the microcode instruction exits the instruction fetch buffer in the same group as the preceding non-microcode instruction. For this reason, there is always at least a one cycle bubble in the decode pipeline when a microcode instruction follows a non-microcode instruction. If the microcode instruction exits the buffer in a separate group (for example, when it is immediately preceded by a branch), the bubble is two cycles.

Certain instructions performed by the microcode pipeline are accelerated by the inclusion of special hardware. This allows strings and multiples to be decoded at up to four load or store operations per cycle. (Some earlier implementations of the PowerPC AS Architecture could decode only one load or store per cycle.) The microcode sequence for string loads and stores, and for load multiple and store multiple instructions, are always followed by a bubble of three cycles in the flow of dispatch groups.

Table 6-7 Microcoded Instructions (No Alignment) lists the original instructions and the internal instructions the microcode decode pipeline generates for instructions with no alignment. *Table 6-8 Microcoded Instruction (Misaligned Loads and Stores)* lists the original instructions and the internal instructions the microcode decode pipeline generates for misaligned loads and stores. See *Table 6-4 Pipeline Bubbles Due to Microcode* on page 171 for a summary of all decoder pipeline bubble conditions.

Table 6-7. Microcoded Instructions (No Alignment) (Page 1 of 2)

Instruction	Pattern
addc.	addc_dcds addc_caoc cmpxi
addco	addc_dcds addco addc_caoc
addco.	addc_dcds addco addc_caoc cmpxi
addeo.	addeo adde cmpxi
addmeo.	addmeo addme cmpxi
addzeo.	addzeo addze cmpxi
divd.	nop_ori divd cmpxi
divdo.	nop_ori divdo cmpxi
divdu.	nop_ori divdu cmpxi
divduo.	nop_ori divduo cmpxi
divw.	nop_ori divw cmpxi
divwo.	nop_ori divwo cmpxi
divwu.	nop_ori divwu cmpxi
divwuo.	nop_ori divwuo cmpxi
lbzux	add lbzx addi
ldux	add ldx addi
lhau	lhz extsh addi
lhaux	lhzx add extsh addi
lhzux	add lhzx addi
lmw	lwz_mul lwz_mul lwz_mul lwz_mul lwz_mul lwz_mul lwz_mul lwz_mul *repeats

Table 6-7. Microcoded Instructions (No Alignment) (Page 2 of 2)

Instruction	Pattern
lswi	nop_ori nop_ori nop_ori nop_ori lsw lsw lsw lsw lsw lsw lsw lsw *repeats
lswx	mfxerf3 add nop_ori lsw lsw lsw lsw lsw lsw lsw lsw *repeats
lwaux	lwzx add extsw addi
lwzux	add lwzx addi
mfcr	mfcrfa mfcrfb or mfcrfc mfcrfd or or
mfspr_xer	mfxerf2 mfxerf0 or mfxerf3 or mfxerf1 or nop_ori
mtcrf	mtcrf_cr0 mtcrf_cr1 mtcrf_cr2 mtcrf_cr3 mtcrf_cr4 mtcrf_cr5 mtcrf_cr6 mtcrf_cr7
mtspr_xer	mtxerf2 mtxerf0 mtxerf1 mtxerf3
mtsr	mtsrdat nop_ori nop_ori mtsradr
mtsrin	mtsrdat nop_ori nop_ori mtsradrin
rlwimi.	rlwinm wi cmpxi
slbia	mtsrdatd nop_ori nop_ori slbia
slbie	mtsrdatd nop_ori nop_ori slbie
slbmte	mtsrdatd nop_ori nop_ori mtsradrslb
stbux	stbx_agen stb_data add
stdcx.	stdcx_agen stdcx_data nop_ori mcrxrf2
stdux	stdx_agen std_data add
sthux	sthx_agen sth_data add
stmw	stw_mul stw_mul stw_mul stw_mul stw_mul stw_mul stw_mul stw_mul *repeats
stswi	nop_ori nop_ori nop_ori nop_ori stsw stsw stsw stsw stsw stsw stsw stsw *repeats
stswx	mfxerf3 add nop_ori stadrchk nop_ori nop_ori stsw stsw stsw stsw stsw stsw stsw stsw *repeats
stwcx.	stwcx_agen stwcx_data nop_ori mcrxrf2
stwux	stwx_agen stw_data add
subfc.	subfcdcds subfccaoc cmpxi
subfco	subfcdcds subfco subfccaoc
subfco.	subfcdcds subfco subfccaoc cmpxi
subfeo.	subfeo subfe cmpxi
subfmeo.	subfmeo subfme cmpxi
subfzeo.	subfzeo subfze cmpxi
tlbie	mtsrdatd nop_ori nop_ori tlbie
tlbiel	mtsrdatd nop_ori nop_ori tlbielcl
tlbiel (L = '1')	mtsrdatd nop_ori nop_ori tlbielpg

Table 6-8. Microcoded Instruction (Misaligned Loads and Stores) (Page 1 of 2)

Instruction	Pattern
ld	lsd addi lsd srd or nop_ori nop_ori nop_ori
ldu	lsd addi lsd srd or addi nop_ori nop_ori
ldux	lsd add lsd srd add or addi nop_ori
ldx	add lsd srd or nop_ori nop_ori nop_ori
lfd	lsd addi lsd srd or nop_ori nop_ori gprtofpr
lfdu	lsd addi lsd srd or addi nop_ori gprtofpr
lfdux	lsd add lsd srd or add nop_ori gprtofpr
lfdx	add lsd srd or nop_ori nop_ori gprtofpr
lha	lbz_s lbz_s slw or extsh
lhau	lbz_s lbz_s slw or extsh addi
lhaux	lbzx_s add lbz_s slw or add extsh addi
lhax	lbzx_s add lbz_s slw or extsh
lhbrx	add lbzx_s lbz_s slw or nop_ori nop_ori nop_ori
lhz	lbz_s lbz_s slw or
lhzu	lbz_s lbz_s slw or addi
lhzux	lbzx_s add lbz_s slw add or addi
lhzx	lbzx_s add lbz_s slw or
lswi	nop_ori nop_ori nop_ori nop_ori lsw lsw srw or lsw lsw srw or *repeats
lswx	nop_ori add nop_ori nop_ori lswx lsw srw or lsw lsw srw or *repeats
lwa	lsw add lsw srw or extsw
lwaux	lswx add lsw srw or add extsw addi
lwax	lswx add lsw srw or extsw
lwbrx	add lbzx_s lbz_s slw or lbz_s slw or lbz_s slw or nop_ori
lwz	lsw lsw srw or
lwzu	lsw lsw srw or addi
lwzux	lswx add lsw srw add or addi
lwzx	lswx add lsw srw or
std	sld stsd addi stsd
stdu	sld stsd addi stsd
stdux	sld stsd_agen stsd_data add stsd add nop_ori nop_ori
stdx	sld stsd_agen stsd_data add stsd nop_ori nop_ori nop_ori
stfd	stfdu_agen stfdb_data stfdl_agen stfdc_data
stfdu	stfdu_agen stfdb_data stfdl_agen stfdc_data addi
stfdux	stfdu_agen stfdb_data add stfdl_agen stfdc_data add
stfdx	stfdu_agen stfdb_data add stfdl_agen stfdc_data
sth	srw stb_s stb_s nop_ori

Table 6-8. Microcoded Instruction (Misaligned Loads and Stores) (Page 2 of 2)

Instruction	Pattern
sthbrx	srw stb_agen_s stb_data_s add stb_s nop_ori nop_ori nop_ori
sthu	srw stb_s stb_s addi
sthux	srw add stb_agen_s stb_data_s stb_agen_s stb_data_s add nop_ori
sthx	srw add stb_agen_s stb_data_s stb_agen_s stb_data_s nop_ori nop_ori
stswi	nop_ori nop_ori nop_ori nop_ori slw stsw stsw add slw stsw stsw add *repeats
stswx	nop_ori add stadrchk nop_ori slw stsw stsw add slw stsw stsw add *repeats
stw	slw stsw stsw nop_ori
stwbrx	stb_agen_s stb_data_s add nop_ori srw stb_s srw nop_ori stb_s srw stb_s nop_ori
stwu	slw stsw stsw add
stwux	slw stsw_agen stsw_data add stsw add nop_ori nop_ori
stwx	slw stsw_agen stsw_data add stsw nop_ori nop_ori nop_ori

6.3.3.3 Run Time Feedback-Based Instruction Cracking

Misaligned address references that cannot be corrected within the data format block of the load store unit[1] result in a flush of the instruction. The instruction is refetched, and the instruction is decomposed into two loads or two stores for each load or store in the original decode of the instruction. In the case of strings, the entire sequence of loads or stores is decomposed in this manner even if some of the instructions previously completed and updated the architected state of the machine.

6.3.3.4 Unused Cycles in the Flow of Dispatch Groups

The following conditions can introduce unused cycles in the flow of dispatch groups from the decoder:

1. If the last instruction in quadword 0 of the instruction buffer and all valid instructions in quadword 1 are included in a group exiting the buffer, then there is one unused cycle.
2. If a microcode instruction follows a non-microcode instruction when exiting the instruction fetch buffer and a or b below is true:
   a. If the microcode instruction is in the same group as the preceding instruction, then there is one unused cycle.
   b. if the microcode instruction is in a separate group from the preceding instruction, then there are two unused cycles.
3. There is one unused cycle for all string load, string store, load multiple, and store multiple instructions.

1. This block has a 64-byte boundary for L1 load hits, a 32-byte boundary for L2 load hits, and a 4-KB boundary for stores.

6.3.4 Instruction Dispatch, Grouping, and Interlocks

6.3.4.1 Resource-Based Instruction Grouping

From dispatch to completion, instructions are tracked in groups. That is, the state of the machine is preserved at the group boundaries, not at the instruction boundary within a group. Any exception will cause the machine to be restored to the state of the oldest group before the exception.

A group contains up to five internal instructions (IOPs). In the decode stages, the instructions are placed sequentially in a group; the oldest instruction is placed in slot 0, the next oldest one in slot 1, and so on. Slot 4 is reserved for branch instructions only. If required, no-ops are inserted to force the branch instruction to be in the fourth slot. If there is no branch instruction, then the fourth slot contains a no-op. Only one group of instructions can be dispatched in a cycle, and all instructions in a group are dispatched together (that is, no partial group dispatch).

Each dispatch slot has an affinity to the issue queue and execution unit as *Table 6-9* shows.

Table 6-9. Decode Slot/Dispatch Restrictions

Slot 0	Slot 1	Slot 2	Slot 3	Slot 4
CRLogical_odd	CRLogical_even			
mtspr_0_odd mfspr_0_odd				
	FXU_divide_1_odd			
FXU_0_odd	FXU_1_odd	FXU_1_even	FXU_0_even	
LSU_0_odd	LSU_1_odd	LSU_1_even	LSU_0_even	
FPU_0_odd	FPU_1_odd	FPU_1_even	FPU_0_even	
				Branch_any
VPERM	VPERM	VPERM	VPERM	
VALU	VALU	VALU	VALU	

In *Table 6-9*, "0" and "1" indicate which execution unit of that type receives the instruction. "Odd" and "even" indicate in which entries of the issue queue the instruction can be stored, based on issue queue restrictions. For example, FXU_divide_1_odd in slot1 indicates that the fixed-point divide operation always goes to an odd entry in the issue queue for the FXU1 execution unit. Similarly, mtspr_0_odd indicates that the Move to Special Purpose Register (**mtspr**) instruction always goes to an odd entry in the FXU0 execution unit.

During dispatch, various machine resources are assigned to instructions. These resources include:

Global completion table (GCT)	One entry per group
Fixed point unit (FXU) issue queue	One entry per fixed-point or load/store IOP
Floating point unit (FPU) issue queue	One entry per floating-point IOP
Branch unit (BRU) issue queue	One entry per branch IOP
Condition register unit (CRU) issue queue	One entry per CR IOP

Mappers	One entry per new destination (either GPR, FPR, XER, CR, LR, or CTR)
Load reorder queue (LRQ)	One entry per load IOP
Store reorder queue (SRQ)	One entry per store IOP
Vector unit arithmetic logic unit (VALU) issue queue	One entry per VALU instruction
Vector permute unit (VPERM) issue queue	One entry per VPERM instruction

These resources are released at various stages:

GCT	When the group completes
Issue queue	After the instruction has been successfully issued
Mappers	When the group completes
LRQ	When the group completes
SRQ	When the store is sent to the STS

All resources for all instructions of the same type (fixed, floating, branch, CR logical, and some special purpose register [SPR] moves) must be available for the group to be dispatched. For example, if there is no odd entry available in the FXU0 issue queue, then a group with at least one instruction that is destined to the FXU issue queue (fixed-point or load/store IOP) will not be dispatched, even if that instruction is not destined for the FXU0 issue queue. However, a group with no fixed-point or load/store IOP is allowed to be dispatched.

Dispatch is also constrained by the following rules:

- An instruction that uses a non-renamed SPR is blocked from dispatch if any instruction that sets a non-renamed SPR instruction has not executed.
- The Synchronization (**sync**), Instruction Cache Synchronize (**isync**), and Lightweight **sync** (**lwsync**) instructions are blocked from being dispatched if any older load IOP has not returned data. They are also blocked from being dispatched if any older store has not gone through the address generation stage (AGEN).

6.3.4.2 Synchronization-Based Instruction Grouping

Several classes of instructions, listed in *Table 6-10* on page 182, force a dispatch group to end. These are instructions that set non-renamed resources, instructions that are required to be the only instruction in a group, and a few special cases for system calls and sync-like operations.

In addition, certain operations are not allowed to execute speculatively in the 970MP processing unit. To ensure that the operation executes non-speculatively, the 970MP processing unit forces them to wait until they are the next to be completed before allowing them to execute. This mechanism is called completion serialization. A side effect of instruction grouping and this serialization mechanism is that the instruction must be dispatched in a single PowerPC instruction group.

Listed below are some cases where this rule applies:

- Operations that update non-renamed resources (such as move to SPRs)
- Load or store operations that accesses guarded (G equals '1') space
- Certain context synchronizing instructions (CSIs) (Return from Exception Doubleword [**rfid**], System Call (**sc**), Move to Machine State Register [**mtmsr**], Move to Machine State Register Doubleword [**mtmsrd**])
- The move to and move from Floating-Point Status and Control Register (FPSCR) floating-point instructions, where the record bit (RC) of the instruction is '1'.
- The move to and move from Vector Status and Control Register (VSCR) instructions

Other conditions force an instruction to be in a single instruction group. Typically, instructions that must be dispatched in a single instruction group are:

- Instructions that require completion serialization
- **lwarx**, Load Doubleword and Reserve Indexed (**ldarx**), Store Word Conditional Indexed (**stwcx**), and Store Doubleword Conditional Indexed (**stdcx**) instructions

Table 6-10. Instructions with Synchronizing Properties

Instruction	Scoreboard		Completion Serialization
	Set	Check	
addc	Set		X
fabs.			x
fadd.			x
fadds.			x
fcfid.			x
fctid.			x
fctidz.			x
fctiw.			x
fctiwz.			x
fdiv.			x
fdivs.			x
fmadd.			x
fmadds.			x
fmr.			x
fmsub.			x
fmsubs.			x
fmul.			x
fmuls.			x
fnabs.			x
fneg.			x
fnmadd.			x
fnmadds.			x
fnmsub.			x
fnmsubs.			x
fres.			x
frsp.			x
frsqrte.			x
fsel.			x
fsqrt.			x
fsqrts.			x

Instruction	Scoreboard		Completion Serialization
	Set	Check	
fsub.			x
fsubs.			x
mcrfs			x
mffs			x
mffs.			x
mfmsr		Check	
mfspr non-renamed SPR		Check	
mfsr		Check	
mfsrin		Check	
mtfsb0.			x
mtfsb1.			x
mtfsf.			x
mtfsfi.			x
mtmsr	Set		x
mtmsrd	Set		x
mtspr non-renamed SPR	Set		x
mtsr	Set		x
mtsrin	Set		x
rfid	Set	Check	
sc	Set	Check	
slbia			x
slbie			x
slbmfee		Check	
slbmfev		Check	
slbmte	Set		x
subfc	Set		×
tlbie			x
tlbiel			x

6.3.4.3 Instruction Grouping Based on Flush Templates

The instruction decode pipeline cooperates with the instruction sequencing unit in breaking up dispatch groups to provide precise exceptions. More specifically, because the instruction sequencing unit tracks instruction progression through the machine in groups, when a particular instruction within a group needs to signal an interrupt this is achieved by flushing all of the instructions (and results) of the group and then re-dispatching the instructions into individual groups. In these cases, the instruction decode pipeline receives a special grouping mask that helps control how the next group of instructions should be packaged.

A similar mechanism is used to ensure that the summary overflow bit in the Fixed-Point Exception Register is correctly maintained.

6.3.5 Register Renaming

The 970MP processing unit performs register renaming on most of the key architectural registers. Renaming serves several purposes and is fundamental to the operation of the inner core. First, renaming allows the 970MP processing unit to manage many of the write after read (WAR) and write after write (WAW) hazards that are present in compiled code. For example, if instruction *n* is going to read R1 and instruction *n + 1* is going to write R1 (a WAR hazard), then, in a non-renamed machine, instruction *n + 1* must wait for instruction *n* to execute before it executes. In the 970MP processing unit, this false dependency chain is broken by renaming logical register R1 to another physical register, R43 for example, so that this conflict of resources does not occur. The renaming logic also ensures that from that point forward all reference to logical R1 are mapped to physical R43.

Register renaming also allows the 970MP processing unit to manage speculative execution, and to be able to quickly recover from mis-speculation. Each cycle, as speculative instructions are dispatched, information about the complete architectural state of the machine is logged in a way that allows it to be recovered in the future, if needed. Rather than saving all of the register state every cycle and for every instruction, the 970MP processing unit uses the larger physical (renamed) register set along with compacted control information to minimize this overhead.

Renaming occurs on the GPRs, FPRs, and Vector Register files (VRFs), the Vector Save/Restore Register (VRSAVE), the VSCR, the Link Register, the Count Register, the Condition Register, the Fixed-Point Exception Register, and the FPSCR. In general, each instruction that is dispatched must have all its source operands and all its target operands renamed. The 970MP processing unit is capable of dispatching up to five instructions per cycle, and each PowerPC instruction can have many source operands and many target operands. Therefore, to minimize the number of rename ports required, many of the more complex instructions are broken up into simpler and more regular internal operations (see *Section 6.3.3 Instruction Decode, Cracking, and Microcode* on page 170 for more information). In addition, although the physical register sets are much larger than the logical register sets, there are circumstances where all of the physical resources are in use. In these cases, the 970MP processing unit is forced to pace instruction progression based on the availability of these resources. *Section 6.3.5.1* through *Section 6.3.5.7* describe the renaming structure of the various architecture facilities.

6.3.5.1 GPR Renaming

The format for internal operations that use or set the GPRs are restricted to two source GPR operands and one target GPR operand. Up to four of these internal operations can be dispatched each cycle (the fifth dispatch slot is reserved for branches only), which leads to the need for renaming eight source operands and four target operands in each cycle.

The PowerPC defines 32 logical GPRs; however, the internal microcode that is used to help emulate some of the more complex instructions defines four additional emulation GPRs (eGPRs). These 36 logical registers are mapped into a bank of 80 physical GPRs. There are two copies of the GPRs, each with four read ports and five write ports.

6.3.5.2 Link and Count Register Renaming

The Link Register and the Count Register, which are two logical registers, are renamed into a common pool of 16 physical registers with three read ports and three write ports.

6.3.5.3 Condition Register Renaming

The Condition Register is logically divided into eight 4-bit fields that can be separately renamed. In addition, the internal microcode that is used to help emulate some of the more complex instructions defines an additional 4-bit field, the emulation CR (eCR) field, which is also subject to CR renaming. This logical set of nine CR fields is renamed into a pool of 32 physical fields with three read ports (one for the branch execution unit and two for the CR logical execution unit) and five write ports (one for the CR logical execution unit, two for the fixed-point execution unit, and two for the floating-point execution unit).

Some aspects of the Condition Register are difficult for a renamed machine. More specifically, the CR logical instructions use CR bits as source and target operands, which requires that these instructions perform a read-modify-write function on the target field (that is, a 4-bit field is read, a 1-bit result is written into one of the bits, and the 4-bit field is written back). As a result, these instructions could require three source rename ports, which is undesirable. Instead, as explained in *Section 6.3.11.1 Condition Register Logical Instructions* on page 199, many of these instructions are cracked so that each internal operation still only requires two rename ports for source operands.

Another difficulty of Condition Register renaming is associated with the Move To Condition Register Fields (**mtcrf**) and Move From Condition Register (**mfcr**) instructions. In their worst form, these instructions require up to eight source or target fields (the whole Condition Register itself). As a result, most forms of these instructions are broken up by the emulation microcode into a series of simpler operations that require a smaller number of rename ports.

6.3.5.4 FPR Renaming

FPR renaming is very similar to GPR renaming except that each internal operation is allowed to have up to three source operands. This allows the multiply-add class of instructions to flow through the machine without being cracked. The FPR rename structure uses a pool of 80 physical registers. Each 970MP processing unit maintains two copies of the FPR, each with three read ports and four write ports.

6.3.5.5 XER Renaming

The XER is architecturally a collection of miscellaneous state bits associated with the execution of various fixed-point instructions. For some, it contains bits that control instruction execution (source operands), and for some it acts as a target area for various exception and state bits. Because some of these bits are somewhat related, and some are used or set more frequently than others, it is possible to group some of these bits into rename fields.The OV and CA bits are renamed from a pool of twenty-four, 2-bit registers.

The summary overflow bit, SO, in the XER is a "sticky" bit that must reflect the overflow status of all instructions before the one that is executing. Instead of creating a stream of dependencies on the SO bit, the 970MP processing unit performs a form of value prediction on the state of the bit. More specifically, the 970MP processing unit assumes that the state of the bit does not change. Then, at completion time, if the execution unit reports that a particular operation really did cause an overflow and the SO bit must be updated, the machine initiates a flush. This causes subsequent instructions to be re-executed, and to pick up and reflect the corrected state of the SO bit when they complete.

In addition, for some bits that are not frequently used, a simple scoreboard function helps to keep track of instructions that update these bits. It helps to prevent instructions that use the bits from proceeding until it is known that previous instructions have completed their required updates.

The format for the internal operations allows up to one XER source field and one XER target field to be renamed per instruction, per cycle. Because up to four internal operations can be dispatched per cycle, this leads to four source operand renames and four target operand renames.

6.3.5.6 FPSCR Renaming

The FPSCR presents a number of problems for an out-of-order machine. They include: updating the CR Register with summary data, maintaining sticky exception information, and updating the control fields needed for proper arithmetic execution. Unlike the FPR rename, the FPSCR rename is coupled with the completion function. There is one FPSCR rename per active group. The FPSCR rename is implemented using a 20-entry buffer that keeps the state of the FPSCR associated with the 20 entries in the global completion table. The following restrictions apply to instructions that affect the FPSCR:

- The following floating-point instructions with RC set to '1' must be dispatched in a single instruction group: **mtfsf**, **mtfsfi**, **mcrfs**, **mtfsb0**, **mtfsb1**, and **mffs**.
- The following floating-point instructions with RC set to '1' require completion serialization before they can be issued: **mffs** and **mcrfs**.

FPSCR rename effects on floating-point instructions are:

- The following floating-point instructions with RC set to '1' execute slowly as they must wait for older instructions to complete: **mffs** and **mcrfs**.
- All floating-point instructions dispatched after a move to the FPSCR instruction (that is: **mtfsf**, **mtfsfi**, **mcrfs**, **mtfsb0**, and **mtfsb1**) must wait for the move to FPSCR instruction to execute before they can execute.

6.3.5.7 VRF, VRSAVE, and VSCR Renaming

VRF renaming allows each vector instruction to have up to three source operands in addition to one source operand for VPU stores. There are 32 architected vector registers, and the rename structure uses a pool of 80 physical registers. Each 970MP processing unit maintains two copies of the VRF, each with four read ports and four write ports.

The VRSAVE Register shares resources with the GPRs. VRSAVE renames are allocated from among the 80-entry register file used to rename the GPRs.

The VSCR has only two of its bits defined. Of these, only the SAT bit, indicating that a vector-saturating type of instruction generated a saturated result, is renamed. The VSCR[SAT] bit is renamed from a pool of 20, single-bit registers.

6.3.6 Instruction Issue and Register File Access

As described above, up to five internal operations can be dispatched and renamed per cycle. After renaming, these operations are placed into a set of issue queues that are distributed by instruction type. There are eight different issue queues, one for each of the following instruction types:

- FXU0/LDST0 (18 entries)
- FXU1/LDST1 (18 entries)
- Branch operations (12 entries)
- CR logical operations (10 entries)
- FPU0 operations (10 entries)
- FPU1 operations (10 entries)
- Vector arithmetic logic unit (ALU) operations (20 entries)
- Vector permute operations (16 entries)

In addition, the FXU0/LDST0, FXU1/LDST1, FPU0, and FPU1 issue queues are partitioned into an even half and an odd half. Fixed-point or load/store operations from slot 0 in the dispatch groups can occupy only the odd half of the 18 entries in the FXU0/LDST0 issue queue. Fixed-point or load/store operations from slot 3 can occupy the even half of the FXU0/LDST0 issue queue. Fixed-point or load/store operations from slot 1 and slot 2 can occupy the odd and even half of the FXU1/LDST1 issue queue, respectively. Similarly, floating-point operations from slot 0 and slot 3 occupy the odd and even half of the FPU0 issue queue, respectively, and the floating-point operations from slot 1 and slot 2 occupy the odd and even half of the FPU1 issues queue, respectively. When any of the issue queues (or any half for the partitioned issue queues) is full, it causes a pipeline stall.

During the issue stage, the issue logic surveys all of the entries of the issue queues and detects which instructions are ready to execute. An instruction is considered ready to execute if it is valid, if all of its source operands are available, and if the various timing and control bits are set in an appropriate state. If multiple instructions are eligible for issue, then the issue logic attempts to issue the oldest instructions in each particular queue. Up to two instructions can be issued from the FXU0/LDST0 issue queue (one FXU operation and one load/store operation), two instructions from the FXU1/LDST1 issue queue (one FXU operation and one load/store operation), two instructions from the branch and CR issue queues (one branch and one CR operation), two instructions from the FPU issue queue (two floating-point operations), and two instructions from the vector issue queues (one vector ALU and one vector permute).

Following the issue stage, the instructions enter the register file access stage where they access their source operands from the various register files that contain the architectural state. A register file corresponds to each of the rename mappers, and the register files are accessed with the renamed address (physical register number).

Some instructions remain in the issue queue for several cycles after they have issued. This is to allow for the possibility of certain types of downstream events preventing the instruction from actually executing, and thereby requiring the instruction to be reissued at a future time (called the reject mechanism).

6.3.7 L1 Data Cache Management

6.3.7.1 L1 D-Cache Indexing and Replacement Algorithm

The L1 D-cache is 32-KB, 2-way set-associative with two read ports and one write port. The replacement algorithm is round-robin and LRU, and it is updated on L1 reload. On cache accesses, effective-address bits are used to index into the L1 D-cache, the directory, and into data ERAT (D-ERAT). The directory contains real addresses that are compared against the real address coming from the ERAT. When these real addresses match, a cache hit has occurred. When these addresses do not match, a cache miss has occurred. The set that was not updated on the previous reload is deallocated when the new reload occurs. It is also important to note that, if multiple effective addresses reference the same real address (aliasing), only one entry is kept in the D-cache at the location indexed by the effective address (EA) of the last of these references.

6.3.7.2 Misaligned Storage References

The LSU performs most loads and stores that are unaligned with the same timing as normal loads and stores with the following exceptions. When these exceptions occur, a special flush is generated to refetch the instruction and cause the microcode to break up the instructions into multiple loads and stores.

- Load crosses a 64-byte boundary on an L1 hit
- Load crosses a 32-byte boundary on an L1 miss
- Store crosses a 4-KB boundary

For simple loads (not string instructions), the group with the unaligned load or store is flushed and re-executed with microcode that avoids the boundary crossing, if the load or store was the first instruction in the group. If not the first in the group, the group is first re-executed with a single instruction per group. The load or store then causes a second flush to generate the unaligned microcode. String instructions are already microcoded so only one flush occurs.

6.3.8 Load Instruction Handling

6.3.8.1 Cache Misses on Loads

The L1 D-cache is physically a true 2-port array capable of two reads and one write per cycle. On a cache miss, data is returned on the L2 cache interface in four 32-byte beats. The L2 always returns the critical sector (the sector containing the specific word address that references the cache line) in the first beat. The load miss queue (LMQ) forwards these demand-oriented loads into the pipeline as quickly as possible, using critical data forwarding (CDF). The other half of the 64-byte L2 read is returned in the second beat. A CDF operation causes one reject slot to be created in the issue pipeline for that LSU port. As each 32-byte beat is received, it is written to the cache. When all four 32-byte beats are received, the D-cache directory is updated. A directory update causes one reject slot on each LSU port (total of two) to be created (this is referred to as AGEN steal).

In addition to loads, **dcbt** and **dcbtst** are placed in the LMQ.

6.3.8.2 Load Reorder Queue

The LSU contains a 32-entry LRQ, which holds real addresses and tracks the order of loads. The LRQ provides the following functions:

- Loads query the LRQ for load-hit-load (LHL) and, when detected, flushes all instruction after the load.
- Stores query the LRQ for store-hit-load (SHL) and, when detected, flushes all instructions after the store.

The LRQ also participates in TLB Invalidate Entry (**tlbie**) instruction processing by marking instructions in the LRQ that must complete before the TLB Synchronize (**tlbsync**) instruction is allowed to complete.

6.3.8.3 Data Address Translation

Effective addresses are first translated into real addresses by a data ERAT or an instruction ERAT. See *Section 6.3.1.6 Instruction Fetch Address Translation* on page 163 for more information about the I-ERAT. The D-ERAT contains 128 entries, each of which provides translation for a 4-KB page. The D-ERAT is 2-way set-associative with an LRU replacement algorithm. The D-ERAT index uses bits 46 - 51 of the effective address (EA[46:51]).

Up to two D-ERAT misses are entered in the ERAT miss queue (EMQ), one for each LSU pipe. These two D-ERAT misses plus one I-ERAT miss arbitrate for the second-level memory management unit consisting of the SLB and TLB. The EMQ entries are the only two D-ERAT misses that are sent to the TLB; others result in a reject. The SLB and TLB are accessed in parallel. If there is a hit in both, for non-guarded accesses, the information for the D-ERAT is returned. TLB hits are handled speculatively, and the I-ERAT and D-ERAT are loaded speculatively in this case. TLB misses are not handled speculatively. When the load/store with the TLB request enters the next-to-complete group, the request is no longer speculative. When the TLB receives a non-speculative request, it can initiate a table walk on a miss. Next-to-complete requests have priority for EMQ entries. A next-to-complete request can steal the EMQ entry from a non-next-to-complete entry. However, if the request has already won the TLB arbitration, it will continue.

The TLB has 1024 entries and is 4-way set-associative. It uses a true least recently used (LRU) replacement algorithm. Each entry in the TLB can contain either a 4-KB page or a large, 16-MB page. For 4-KB page addresses, the TLB is indexed with EA[44:51]. The SLB and TLB are accessed in parallel; the TLB access is initiated assuming a 4-KB page. If the SLB entry indicates the address is part of a large page, the TLB is reaccessed, this time indexed for a large page with VA[33:40]. A large-page TLB hit is two cycles longer than a 4-KB page hit, because of this reaccess.

If the instruction is a load, the secondary page table entry groups (PTEG) search starts after the primary PTEG search has failed. If there is a page fault, an exception will be signaled to the instruction sequencer unit (ISU) if the load is in a single-instruction group or part of a microcoded group. If if the load is not in a single-instruction group, the load is flushed to create single-instruction groups. The entire process, including the tablewalk, is repeated.

If the instruction is a store, the flush to create a single-instruction group happens if there is no match in the primary PTEG or if there is a match but the C bit is not set. If this flush happens, the entire execution of the store repeats. This time, the secondary PTEG is searched after the primary PTEG has been searched. If the store was originally in a single-instruction group, or was part of microcode, the secondary PTEG would have been searched after the primary search failed. During the entire tablewalk, the load/store is continuously rejecting (except while the instruction is flushed). When the D-ERAT is loaded, there may be some cycles before the instruction can continue, depending on where in the reject cycle it is. During a TLB miss tablewalk, no other ERAT misses can access the TLB.

Note: If the load or store is not in a single-instruction group, there could be multiple TLB misses within the group. The tablewalks could occur in any order (but one at a time), depending on which gets to the TLB first after becoming next-to-complete. For a load instruction, the reference (R) bit in a page table entry is set during the tablewalk. A previous instruction in the group could generate an exception after the R bit has been set. For a store that crosses a page boundary, either a store multiple/string or an unaligned simple store, once next-to-complete, the second page tablewalk could happen before the first page tablewalk. Therefore, the change (C) bit on the second page could be set before the first page tablewalk. When a page is initially loaded in the D-ERAT with the C bit off, a store to the page will result in a D-ERAT miss. If the TLB has the C bit off, then there will be a TLB miss and tablewalk to set the C bit.

6.3.8.4 Loads in Real Mode

For load instructions in real mode (MSR[DR] equals '0' or G equals '1'), several different scenarios are possible. First, if the load instruction is the next instruction to complete, then it is considered non-speculative; if the load instruction is not the next-to-complete, it is considered speculative. In either case, when the real-mode I bit located in HID4 is not set, the load is in cacheable mode. In cacheable mode if there is an L1 D-cache hit, the load can access any level of the memory hierarchy whether the load is speculative or not. If there is a hit in the L1 D-cache, it is known that the line has been referenced before, and so it is safe to execute the load even speculatively. Therefore, if there is an L1 D-cache hit, the load in real mode behaves like any other normal load from a performance perspective. Real mode references to the D-ERAT are always considered hits even though no entry is created in the D-ERAT.

If there is a miss in the L1 D-cache for a load in real mode, then it may not be safe to execute the load speculatively. If the load is not in its own group, a flush occurs to form single-instruction groups. Once in a single-instruction group, the load is rejected until it is non-speculative. When non-speculative, the load is treated as a normal cache miss. When the I bit is set (cache-inhibited mode), the same basic things happen except that by definition there will always be cache misses.

Note: As a result, the load action can only be detected by memory-mapped I/O once per load instruction.

Guarded pages (G equals '1') are installed in the D-ERAT and TLB. For a D-ERAT hit with G set to '1', the access is the same as for real mode described above, with an L1 D-cache hit or miss determining whether the access can be speculative or non-speculative. For a D-ERAT miss, the miss can proceed speculatively as long as there is a TLB hit. With a TLB hit, the G bit is known and the L1 D-cache hit or miss status determines whether the instruction can be handled speculatively. For a TLB miss, it is handled non-speculatively as are other TLB misses.

6.3.8.5 Round-Trip Load Processing

The processing for load instructions begins at instruction dispatch. In addition to other requirements, dispatching a group requires that four LRQ entries are available, even if there are no loads in the group. Each load operation is installed in one LRQ slot and in one fixed-point issue queue slot. The load issues when its operands are about to become available. Loads execute in one of the two LSU pipes. The pipeline consists of several stages: issue, register read, address generation (AGEN), cache access, data format (align), and register write/finish. A dependent fixed-point instruction will issue three cycles after the load issues, or a dependent floating-point instruction issues five cycles after the load issues (assuming all other conditions are also met). Thus, the load data can bypass the register file for immediate use by the dependent instruction. Similarly, a dependent VPU permute operation will issue four cycles after the load issues, and a dependent VPU ALU operation will issue five cycles after the load issues. If the load encounters certain problems as it

executes, it can cause the load to be rejected. In this case, the load remains in the issue queue and issues again, no sooner than seven cycles after the previous issue. See *Section 6.3.10.1 Load/Store Reject Mechanism* on page 196 for a complete list of reject cases. An issued dependent instruction is also rejected.

When the load effective address has been computed, it is sent to the D-ERAT and L1 D-cache. If the page is not found in the D-ERAT, the load is rejected and will continue this pattern until the D-ERAT entry is loaded, resulting in a hit. In the event of a TLB miss, the load instruction continues to be rejected. A typical D-ERAT miss/TLB hit requires two rejects, or 14 cycles. While processing a D-ERAT miss, the TLB request is entered in the EMQ, if space is available. There is one EMQ entry for each LSU pipe. Rejects continue regardless of finding space in the EMQ. Once the TLB is accessed, on a hit, the real address is installed in the D-ERAT. On a TLB miss, the tablewalk is not initiated until the load instruction becomes non-speculative; that is, it is in the next-to-complete group. If the load does not occupy an EMQ slot when it becomes next-to-complete, it can steal a slot from a non-next-to-complete load. A next-to-complete load cannot have its EMQ stolen.

Once the address is found in the D-ERAT, the real address from the D-ERAT is compared to the real address from the L1 D-cache directory. If they do not match, it is a cache miss. In this case, the load will attempt to acquire an LMQ entry. If the real address does not already exist in the LMQ, and there is an available LMQ, and there is no entry in the LMQ for the same L1 D-cache congruence class, an entry in the LMQ will be created corresponding to this load that includes the real address, length, sign extend, and GPR or FPR number. There is enough information to allow the data to come from the STS and write the register without requiring the load to execute again. Consequently, the load is removed from the issue queue.

Each LMQ entry can hold two separate loads from the same line. If an LMQ entry already exists, but the second load slot is not used, the load information is entered in the existing LMQ entry and the load is removed from the issue queue, as in the case when an LMQ entry is first allocated. If both LMQ slots are occupied, any other loads to the same line will result in a load-hit-reload (LHR) reject. When the data returns from the STS, data for the corresponding load in the LMQ is written to the register (critical data forwarding). This happens for both loads in an LMQ entry. In addition, the STS indicates early that the data is coming. This allows a dependent instruction to be issued just in time to receive the data for either of the loads, assuming other issue constraints are also met. There are cases where the data coming signal is false (such as an L2 cache miss). In this case, the dependent instruction will be rejected and it will remain in the issue queue until the next data coming signal. For loads that had an LHR reject, rejects will continue until the line is marked valid in the L1 D-cache (four cycles after the whole line is received), at which time there will be a cache hit. If a load reaches the address generation stage just after its sector comes in, it will also be rejected until the entire line is written. When a line is loaded in the L1 D-cache, the RaTag array within the LSU is checked for any alias of the line; that is, it checks for different effective addresses (EAs) mapped to the same real address (RA). If an alias is found, the line is invalidated. If two load instructions, executing simultaneously, both miss the L1 D-cache, then both are entered in the LMQ and both requests are sent to the STS the same cycle. This will not happen when the two loads access the same congruence class, or in similar LMQ reject cases.

For a load to succeed, there are a number of other conditions that are checked. If the load is preceded by a store to the same address, store forwarding may occur (see *Section 6.3.9.1 Store Queue and Store Forwarding* on page 193). If the load is not contained within a preceding store, such as a store byte, load half-word sequence, the data cannot be forwarded to the load because part of the data is produced by the store and part from the L1 D-cache. If the load and store are in different groups, the load is rejected until the store writes to the L1 D-cache. If they are in the same group; because the store cannot complete until its entire group has finished, and the load cannot finish without its data; the group is flushed to create single-instruction groups. Now that the store and load are in different groups, the load can be rejected.

Another scenario is when a dependent load is executed before a preceding store has reached the address generation stage. In this case, when the load executes, it is unknown that the load depends on the store. When the store reaches the address generation stage, it checks the LRQ and detects this case. Because the load has left the issue queue, it must be flushed to now execute after the store.

The LRQ includes a mechanism to maintain sequential load consistency. A load-hit-load (LHL) occurs when two loads to the same address execute out-of-order and there is a store from another processor to that address. When an L1 D-cache line is invalidated on a snoop, any loads in the LRQ from that line have a bit set to indicate this. When a load executes, if there is a later load that has already executed such that this bit is set, all entries after the current load are flushed. If the current load is not in a single instruction group and is not the last non-branch instruction in the group, then the current load is flushed and will be forced into a single instruction group. On this next attempt, if the LHL occurs, the groups after the current load are flushed.

Once the load is in the LMQ, or the load gets forwarded data from a store, or the load gets data from the L1 D-cache, the load is removed from the issue queue. If the load has not been flushed, when the load's group completes, the load is removed from the LRQ, and renamed registers become the architected state.

There are some cases where load miss requests are made to the STS, but are then canceled in the following cycle. This is to allow the request to go out faster, but requires recovering from some cases where the request should not have been made. The cases are a D-ERAT hit and an L1 D-cache miss where the instruction is rejected. Some examples are an LHR reject, a same-congruence class reject, or an LMQ full reject.

Load Processing in the CIU and L2

There are enough load queues in the core interface unit (CIU) to hold all of the eight possible LSU load requests. The CIU sends these loads to the L2, given there are no address or resource conflicts. The load requests are handled by one of the six L2 read/claim state machines. If the line is in the L2, the read/claim state machine returns the data (critical double quadword first) through the reload bus to the LSU. If the line is not in the L2, the read/claim state machine initiates a load of the line from memory or another processor's cache using a read bus command. The replaced line in the L2 is chosen by a true LRU algorithm, and will result in a bus cast out operation if the replaced line was modified. As the bus returns the read data, this data is forwarded to the core a double quadword at a time. When the entire line is available, the line is written into the L2.

Load Multiple and Load String

Because Load Multiple and Load String instructions typically load multiple registers, these instructions are microcoded. For Load Multiple, the microcode consists of one load operation for each register loaded. There are four load operations per group, except the last group may be padded with no-ops. Each load operation requires one LRQ entry. The load operations can execute in any order. When a group consists of four load operations, two will be queued for each LSU pipe. In this way, two load operations can execute in each cycle. Many Load Multiple or Load String instructions may require multiple groups of internal operations. Each group completes in order, without waiting for later groups to finish. Consequently, if the instruction is flushed, all groups including those that have completed, are redone.

Load String is much the same as Load Multiple, with a few differences. For Load String Immediate, the first group contains no-ops. After that, there are up to four loads per group. Because the last register may be only partially loaded, the instruction set for internal operations includes instructions to load one to seven bytes (left aligned per load string architecture); both word and double word strings are supported. This results in exactly one load operation per register.

Load String Indexed instructions depend on the string length from the XER. Because the length is needed at the time the microcode is generated, the microcode generation must be delayed. This is accomplished by beginning the load string microcode with an XER read instruction of the string length. This will stall in dispatch until the string length field has been written by a preceding instruction. Also in the first microcode group is an add to compute the starting address. An immediate operand is then added to this address to compute the load address for each operation after the first. This first group ends with no-ops. Subsequent microinstruction groups, consisting entirely of no-ops, are generated so that the decode pipeline is filled while waiting for the length. Once the string length is available, it is sent to the decode unit, which can then create the microcode for the string instruction.

Unaligned string instructions occur when any individual register load crosses a 32-byte or 64-byte boundary. When this happens, the entire string instruction is flushed. Then, each original load operation is replaced by two loads, a right shift, and an OR. The loads take advantage of the capability to specify a load of any number of bytes, from one to seven. The resulting microcode for Load String Immediate is one group of no-ops, then one register loaded per group. For Load String Indexed, there is an address add in the first group, then no-ops. There is no need to wait for XER because that value is known after the unaligned flush.

Load and Reserve

The **ldarx/lwarx** (**larx**) instructions are not microcoded or split, but must be in their own group. Only one **larx** is allowed at a time. A later **larx** is rejected until an earlier one completes and frees its LRQ entry. An **larx** can execute speculatively, but a later **larx** is prevented from executing until it is next-to-complete or there is a store conditional (**stcx**) that executes between it and the previous **larx**. If there are two **larx** instructions and the later **larx** executes first, when the earlier one executes it detects the later one in the LRQ and will flush all groups after itself to clear out the later **larx**.

Because the reservation is located in the STS, all **larx** instructions are forced to miss the D-cache. This results in an L1 D-cache miss, which requires an LMQ slot, and so on. If the line was already in the L1 D-cache it remains. If the line was not in the L1 D-cache, it is not installed. Because the **larx** sends a special request to the STS, there are various interactions with other loads or stores that are different than normal load processing.

In the case of a load and an **larx** to the same line, when the line is in the L1 D-cache, the load is a hit and can get data from the L1 D-cache. The **larx** goes to STS as normal. The program order and execution order do not matter. If the line is not in the L1 D-cache, because the **larx** sends a special request to STS, the **larx** cannot be installed in an existing LMQ entry. An LMQ entry created by an **larx** is not allowed to be shared with another load (and the LMQ can have at most one entry from each L1 D-cache congruence class). Regardless of the execution order between a load and an **larx**, the later one to execute is rejected until the LMQ of the first instruction is freed. Any problems with out-of-order execution are handled in the typical way. For example, should another processor store to the line, the load-hit-load check will enforce the program order. A store instruction cannot store-forward data to an **larx**.

6.3.9 Store Instruction Handling

6.3.9.1 Store Queue and Store Forwarding

The LSU contains a 32-entry SRQ, which holds real addresses and a 32-entry store data queue (SDQ) that holds a double word of data. Store addresses are loaded into the SRQ when the instruction is issued. These instructions can be issued in any order. Fixed-point data is loaded into the SDQ from a dedicated bus from the FXU after the data is accessed from the GPR. Floating-point data is loaded into the SDQ from a dedicated bus from the FPU after the data is accessed from the FPR. Stores are removed from the SRQ and SDQ and written to the cache in program order after all the previous instructions have been committed.

Loads that are issued that hit (address match) stores in the SRQ that have not been written to the cache are candidates for store forwarding. A store forward can occur under the following conditions: the data is available in the SDQ, the load is completely contained within the store (load byte count is less than or equal to the store byte count), the store is older than the load, the page-table I bit or real-mode I bit (located in HID4) is not set, and there is no collision with an L1 reload operation (AGEN steal). If the above conditions are not met, the load instruction can be either rejected or flushed. A load that becomes a store forward operation is treated as a miss from the standpoint of the issue logic. The latency of a store forward operation is six cycles.

In addition to stores, **sync, lwsync,** Page Table Entry Synchronize (**ptesync**)**,** Enforce In-Order Execution of I/O (**eieio**)**,** Data Cache Block Set to Zero (**dcbz**)**,** Data Cache Block Flush (**dcbf**)**, icbi, tlbsync,** and **tlbie** (non-local) are installed in the SRQ.

6.3.9.2 Cache Misses on Stores

The L1 D-cache is a store-through design. As such, when a store is removed from the SRQ, if it hits in the L1 D-cache, the line is updated. In addition, all stores (hit or miss) are forwarded on to the L2. If the store is a miss in the L1 D-cache, the data is not written to the D-cache and the line is not reloaded from the L2 cache (no L1 D-cache allocate on store miss).

6.3.9.3 Store Gathering

Store gathering is not supported in the store queue above the L1 D-cache. It is supported, however, in the store queues below the L1 D-cache and above the L2 cache (in STS).

6.3.9.4 Stores in Real Mode

For store instructions in real mode (MSR[DR] equals '0'), the store queue above the caches is used as a temporary holding spot for both the address and the data. When the store passes the completion point, the store queue is marked to allow the store data to be written into any of the appropriate caches. If the real-mode I bit (located in HID4) is set, we would expect cache misses, so the store works its way towards memory or memory-mapped I/O.

Note: As a result, the store action will only be observed once per store instruction.

6.3.9.5 Round-Trip Store Processing

The processing for store instructions begins at instruction dispatch. In addition to other requirements, to dispatch a group requires that four SRQ entries are available, even if there are no stores in the group. Each store operation is installed in one SRQ slot and in one fixed-point issue queue slot. Most stores issue twice (dual issue). The store address generation (st-AGEN) part executes in one of the two LSU pipes and performs the address calculation. The real address is entered in the store's SRQ entry, once the store completes translation. The store data (st-data) part executes in one of the two fixed-point units, reads the data from the GPR/FPR, and enters it in the SDQ. Some microcoded stores have explicit st-AGEN and st-data parts.

The st-AGEN executes much like a load instruction in that the D-ERAT and TLB are checked. Like a load, a D-ERAT miss will result in a reject. A store differs from a load in that the data cache is not checked during the st-AGEN execution. Store data is only written to the caches when the store completes with its group. The state of the cache may be different at that time. The st-AGEN checks the LRQ for store-hit-load (SHL). A SHL occurs when the program order is a store then a load to the same address, but the load executes first. In this case, the load receives the wrong data because it gets the data before the store takes place. Instructions after the store are flushed (as are load flushes: flush after store if single instruction group or if last non-branch; otherwise, first flush to get a single instruction group). The dependency relationships on previous instructions that write the store's AGEN registers and Data Register are different. Because the data dependency is typically closer to the store than the address dependency, the st-AGEN and st-data are separated. This allows the st-AGEN to typically execute first and can help avoid SHL or other traumas without waiting for the st-data, which would often be delayed.

When the store's group is next-to-complete and all operations have finished, then the stores in the group can be committed to memory. This happens in program order as maintained by the SRQ. The L1 D-cache is checked for hit or miss using the RaTag array (the store address was translated during st-AGEN execution). The RaTag array maintains a real-to-effective mapping of what is in the L1 D-cache. This is used for stores, for alias detection on cache loads from STS, for snooping, and for other miscellaneous functions. By using the RaTag array, the store avoids using the L1 D-cache directory, which could be in use by two loads or stores. If the store's real address (RA) is present in the L1 D-cache, the store writes the data.

In addition to the L1 D-cache, the store data is sent to the STS. While two stores can execute simultaneously due to the two LSU pipes, the store data from only one store can be written to the L1 D-cache and to STS in each cycle. In addition, an individual store that crosses an 8-byte boundary is sent in two pieces to the STS (only VPU stores use the full 16-byte bus). The store also checks the LMQ for the store-hit-reload (SHR) case. This is a store to a line for which a miss is pending. To make sure that the L1 D-cache copy of the store's location is consistent, the store prevents the LMQ from writing the line to the D-cache. A bit is set in the LMQ to do so. When the miss data returns from STS, the one or two loads in the LMQ entry get their data written to the registers (if the load address range overlaps the store address range, this is handled with the typical mechanisms of store forwarding, rejecting, or flushing). Because the line never appears in the L1 D-cache, another load that has been rejected for LHR, will now find that it is a cache miss.

Once the L1 D-cache is written (if a hit) and the STS accepts the store data, the SRQ for the store is freed. This is at best seven cycles after the GCT entry is released. Because of this delay, SRQ entries live longer than all other out-of-order queues.

Store Processing in the CIU and L2

Cacheable stores from the LSU are transferred through CIU store queues into the L2 cacheable store queue, which consists of eight 64-byte buffers. The CIU queues handle vectoring of LSU store commands to either the L2 or the non-cacheable unit (NCU). They also provide flow control when the downstream queues are full. Store gathering is supported in the L2 store queue, with a gather window implemented to optimize gathering. Stores are gathered into a store buffer with a matching 64-byte block address until the buffer arbitrates for a store into the L2 cache. The gather window timer is defaulted off, but can be used to allow more gathering time by delaying the L2 store access, if this proves to provide better performance.

The L2 store queue entries arbitrate in order for an L2 read/claim state machine. Given no address or resource conflicts, the read/claim state machine accepts the store and checks the cache directory (and reads the data as well). If the directory is exclusive (E) or modified (M), the store data is combined with the read data, based on the store buffer byte marks, indicating which data is changed, and the result is written back to the cache. If the directory is shared (S) or recent (R), a data line claim (DCLAIM) bus operation must be completed before the store can complete. If a line is not in the L2, a read with intent to modify (RWITM) bus operation is done to obtain the line data and permission to write the line. In all cases, the modified line ends up in the L2, with the directory marked M.

Non-Cacheable Load, Store, and Instruction Fetch

Non-cacheable load, store, and instruction fetch operations are performed through the NCU. See *Section 3.5.3 Non-Cacheable Unit* on page 98 for additional information.

Store Multiple and Store String

Store Multiple and Store String are processed much like Load Multiple and Load String. Each of these instructions is microcoded with one store operation generated for each register. In a Store String, only some bytes of the last register can be stored. This is also handled with a single operation of one to seven bytes. For Store String Immediate, the first microcode group contains only no-ops. Each of the store operations will result in a st-AGEN issue and a st-data issue. Each operation uses one SRQ entry. The store operations can execute in any order, although they will be written from low to high address as maintained by the order of the SRQ entries. When a group consists of four store operations, two will be queued for each LSU pipe. In this way, two store operations can execute in each cycle. Many Store Multiple or Store String instructions may require multiple groups of internal operations. Each group completes, in-order, without waiting for later groups to finish. Consequently, if the instruction is flushed, all groups, including those that have completed, are redone. This means that these stores could be seen more than once by the STS.

Store String Indexed instructions are processed like Load String Indexed. The first microcode group contains an XER read and an address add. The next several microcode groups are entirely no-ops to fill the decode pipeline. At this point, the length is known and the proper number of stores is generated with up to four stores per group.

Unaligned string instructions occur when an individual store crosses a 4-KB page boundary. This causes the entire instruction to be flushed to be decoded again to avoid the boundary crossing. Each original store now is placed in its own group, consisting of a left shift, two stores, and a no-op. The two stores will each store from one to seven bytes of data left aligned in the register. The first microcode group consists of no-ops. For Store String Indexed, the first group contains an address add. There is no need to wait for XER in the unaligned case.

Store Conditional

The **stdcx/stwcx (stcx)** instructions are microcoded instructions consisting of an **stcx** AGEN operation, an **stcx** data operation, a no-op, and a move from XER to CR. Like all stores, the store waits until it is next-to-complete before a request is sent to the STS. The **stcx** does not store data to L1 at completion, but forces the L1 line, if it is in the L1 D-cache, to be invalidated. When the result is returned from STS, it is written to the XER. The final operation is to copy the final CR value over to the CR. An **stcx** is unlike other stores in that it does not finish as part of execution, but must wait for an STS response. Thus, the finish report sent to the completion unit can cause another LSU operation to be rejected because it cannot send its finish report (**stcx** acknowledgment collision). The LSU tracks that there is a reservation, using a single bit.

If an **stcx** executes without a preceding **larx**, the LSU sends this fact to STS with a new operation, **stcxf**, for which STS always returns a fail.

Data cannot be store-forwarded from an **stcx** to any load.

6.3.10 Storage Access Delays

6.3.10.1 Load/Store Reject Mechanism

The LSU can be issued up to two instructions per cycle on its two ports. The LSU detects several internal conditions that cause it to reject an instruction on a given port. When the LSU rejects an instruction, it causes a 7-cycle turnaround for the instruction to be reissued. *Table 6-11* lists the conditions that cause an instruction to be rejected.

Table 6-11. Conditions That Cause Instruction Rejection (Page 1 of 3)

Functional Block	Condition	Description
Exec	CDF reload collision with issue	When the critical data arrives from the STS, it is formatted and immediately forwarded, bypassing the D-cache, to the GPR (or FPR) using the result bus. Because the result bus is shared by GPR and FPR, any **ld**/**st** instruction that requires the result bus in the same cycle when the CDF is being forwarded is rejected. The reject signal is generated in the finish cycle.
Exec	**stwcx/stdcx** acknowledgment (ACK) collision with issue	When the STS acknowledges the **stwcx**/**stdcx** instruction, it sends the instruction tag (itag) on the control bus of the LSU (side 0). If there is a **ld**/**st** instruction that needs the control bus in the same cycle that STS sends the ACK back, the **ld**/**st** instruction is rejected.
Exec	**mfspr** (second time through pipe) collision with issue	The **mfspr** instruction is processed through the pipe twice; the first time to read the SPR and the second time to get the rename tag and forward the data. If there is a contention for the result bus when **mfspr** is processed through the pipe the second time with another issue, the issue is rejected.
Exec	Store forward (second time through pipe) collision with issue	
Exec	tag/D-ERAT update reload collision (AGEN steal)	If a **ld/st** instruction requires access to either the D-cache directory or the D-ERAT in the same cycle when they are updated, the **ld/st** instruction is rejected. There are two copies of the D-cache directory and the D-ERAT.
Exec	Load speculative, and I equals '1', and single instruction group	The architecture does not allow loads in the cache-inhibited mode (I equals '1') to be speculative. A load from cache-inhibited space waits until it becomes the next-to-complete instruction before it is executed.

Table 6-11. Conditions That Cause Instruction Rejection (Page 2 of 3)

Functional Block	Condition	Description
Exec	Data prefetch collision with issue	L1 data prefetch is generated from the LSU (side 0). Before the prefetch request is sent to STS, the D-ERAT and D-cache directory is checked to see if the line being requested is already in the cache or not. The prefetch request is sent only if the line is not in the D-cache. The prefetch operation has higher priority over other **ld/st** instructions that need to access the D-cache directory and D-ERAT; if there is a collision, the **ld/st** instruction is rejected.
Exec	ERAT miss	If a **ld/st** instruction misses the D-ERAT, it is rejected. The ERAT miss request is sent to the TLB. If there is a TLB hit, the translation data comes back from the TLB in nine cycles. Because a rejected instruction is reissued at least seven cycles after it is rejected, it is likely that an instruction that misses the D-ERAT may end up having two successive rejects.
Exec	Reload critical data forward and **mfspr** (second time through pipe) collision	The CDF has higher priority, and so the **mfspr** is rejected on collision.
Exec	Reload critical data forward and store forward (second time through pipe) collision	The CDF has higher priority, and so the load instruction that needs its data to be forwarded from the store data queue is rejected.
Exec	**stwcx/stdcx** ACK and **mfspr** (second time through pipe) collision	The **stwcx**/**stdcx** instruction has higher priority, and so the **mfspr** gets rejected.
Exec	**stwcx/stdcx** ACK and store forward (second time through pipe) collision	The store forward is rejected when such a collision occurs.
Exec	Speculative load, and guarded (G equals '1'), and single instruction group, and ERAT hit but L1 miss	
LMQ	LMQ full (no entries available)	Load miss queue has eight entries. If all the eight entries are full, subsequent load instructions are rejected.
LMQ	LHR full (no LHR slots available in matching entry)	Each entry in the LMQ has two load-hit-reload (LHR) slots. The LHR slots can be used by two load instructions that miss the D-cache for the same cache line. If more than two load instructions miss the D-cache for the same cache line, the third and subsequent loads are rejected. Critical data forwarding is performed for both the loads in LHR. If there is no queueing delay, it takes four cycles to receive the 128 bytes of a cache line (in four successive cycles) from the STS. It takes an additional two cycles to validate the cache line in the D-cache directory. Because a D-cache hit takes 10 cycles, it takes 15 cycles for the D-cache line to be validated after a cache miss (assuming there is no queueing delay). If a load misses the data back from L2, then it cannot access the data until the entire line is written in the D-cache and the directory is validated.
LMQ	LHR EA alias (no EA match but RA match)	The load miss queue is RA-based. If two loads have the same RA but a different EA, then the second load is rejected.
LMQ	LHR outstanding congruence class request (EA match but no RA match)	Only one D-cache miss is allowed per congruence class.
LMQ	**lwarx/ldarx** outstanding	Only one **lwarx**/**ldarx** is allowed to be outstanding, so subsequent **lwarx**/**ldarx** instructions are rejected. Due to out-of-order issue, it is possible for a younger **lwarx** to be outstanding causing the older **lwarx** to be rejected. If the two **lwarx** are for the same address, when the older **lwarx** is issued without being rejected, it will cause a flush of the younger **lwarx**.
LMQ	LS0_CCA equals LS1_CCA in same cycle	If EA bits 50:56 are the same for misses from LSU0 and LSU1, then the two loads are for the same congruence class and one of them is rejected.
SRQ	Load-hit-store same cycle (store is older)	The younger load will be rejected.
SRQ	Load-Hit-Store (different group, store forwarding not possible)	The load is rejected if the data needed by the load is not fully contained in the SDQ for the store instruction.

Table 6-11. Conditions That Cause Instruction Rejection (Page 3 of 3)

Functional Block	Condition	Description
SRQ	Load-Hit-Store (data not available)	The load is rejected because data to be stored by the store instruction is not yet available in the SDQ.
SRQ	Load-Hit-Sync	The load is younger and so rejected because **sync** has to complete first.
SRQ	Load-Hit-DCB (**dcbz**, **dcbf**)	The load is younger and so rejected because the DCB (**dcbz**, **dcbf**) instruction has to complete (**dcbz** could have forwarded '0', but the 970MP microprocessor does not do that).
SRQ	**lwarx/ldarx** not next-to-complete	If a previous **lwarx**/**ldarx** instruction set the reservation, then a successive **lwarx** is rejected, unless it is next-to-complete. This prevents excessive loss of reservation and still guarantees forward progress.
Global	Force reject for hang detect	When the pervasive unit sends the hang detect signal, all load/store instructions are rejected by the LSU.

6.3.10.2 Flushing Storage Access Conflicts

The LSU detects several internal conditions that cause a flush. This is signaled to the ISU during the finish cycle. *Table 6-12* lists the conditions.

Table 6-12. Conditions That Cause Flushing (Page 1 of 2)

Functional Block	Condition	Description
Exec	I equals '1' and not a single instruction group	Load/store instructions in the I equals '1' space need to be in a single instruction group. This requirement ensures that the instruction is not executed more than once due to flush or reject. A **ld**/**st** instruction in the I equals '1' space is executed when it is in a single instruction group and the instruction is the next to complete.
Exec	G equals '1' and not a single instruction group	All load/store instructions in guarded (G equals '1') storage need to be in a single instruction group. Otherwise, a flush occurs with a special signal that indicates formation of a single-instruction group immediately following the flush.
Exec	Synchronous flush for a speculative load that has a matched snoop in the LRQ	At dispatch time, a **sync** takes up an entry in the SRQ, and a **load** takes up an entry in the LRQ. Because instructions are executed out-of-order, a younger **load** may be executed before an older **sync** instruction. Because the **sync** instruction is older, the younger **load** does not get committed until the **sync** is done with respect to all other processors in the system. If at any point after the execution of the **load** instruction, another processor stores in the storage location from which the data is loaded by the **load** instruction, a bit in the LRQ is set to indicate the snoop match. When completion of the **sync** instruction comes from the STS, the first group with a snoop-matched load instruction is flushed (along with all subsequent groups) to make sure that the load is performed after the **sync** instruction with respect to all other processors. If a younger **load** is executed out-of-order with respect to an older **sync**, but there is no snoop match of the **load** instruction when the **sync** completes, then there is no need to flush the **load** because the effect of out-of-order execution of the **load** is the same as if it was performed after the **sync** with respect to all other processors.
SRQ	Load-Hit-Store EA alias (no EA match but RA match)	A younger **load** hits an older store that is already in the SRQ. However, because there is no EA match, data was not forwarded from the SRQ and stale data may have been forwarded from the D-cache for the load instruction. So the **load** instruction needs to be flushed.

Table 6-12. Conditions That Cause Flushing (Page 2 of 2)

Functional Block	Condition	Description
SRQ	Load-Hit-Store (same group, store forwarding not possible)	If store forwarding is not possible, part of the data to be loaded is in the store data queue (SDQ) and the rest is in D-cache. Because the load and the store are in the same group and only the store can complete and the load cannot, the 970MP processing unit generates a flush to get the load and store instructions in separate groups.
LRQ	Store-Hit-Load	The younger **load** has been executed before the older store is executed. If the two instructions have overlapping data, then the **load** has received stale data, and so it has to be flushed.
LRQ	Load-Hit-Load	If two loads are executed out-of-order, and they have byte overlap (based on the real address), and there is a snoop in between an overlapped byte, then sequential execution consistency might not be preserved. The younger **load** is flushed.
LRQ	**lwarx**-hit-**lwarx**	If a younger **lwarx** is executed before the older **lwarx**, then the younger **lwarx** is flushed. This may happen independent of LMQ detection of an outstanding **lwarx/ldarx**.
LMQ	sequential load consistency flush	If there is a load outstanding for a missed D-cache line, and a snoop to the same line is detected, then the LMQ is marked to indicate that the load should be flushed when the cache line comes back. Because the cache line sent by the STS may be stale due to the snoop, it is safer to do the flush.
LMQ	STS ECC error	After the STS returns the data, it can send an error checking and correction (ECC) error. Because the data has already been accepted by the LSU by the time the ECC error is notified, it will cause a flush.
	Misaligned storage references	See *Section 6.3.7.2 Misaligned Storage References* on page 187.

6.3.11 Handling Other Core Instructions

6.3.11.1 Condition Register Logical Instructions

All Condition Register logical instructions are fully supported in hardware. Each 970MP processing unit allows two CR rename ports, so at most two fields can be read in a given cycle. Where each of the operands is a unique CR field, which are referred to as non-destructive CR logical operations, three CR fields need to be read. Two fields are needed for the two sources and one for the target in which 3 bits should be left unchanged. This cannot be done in one cycle. The 970MP processing unit will automatically crack these instructions into two internal operations. This simplifies the dataflow of the inner core. Where the second operand is the same as the target operand (that is, BT[1] and BB[2] are both in the same CR field), the 970MP processing unit does not crack the instruction. The instruction flows through the machine in a more optimal way.

1. Field used to specify a bit in the CR or in the FPSCR to be used as a target.
2. Field used to specify a bit in the CR to be used as a source.

For CR logical operations where BT and BB are in different CR fields, instruction decode logic creates two internal operations and uses a (non-architected) emulation CR field to store the intermediate result. The first operation reads the two source operands, performs the logical operation, and stores the result in an eCR field. The second operation copies from the eCR field to the target CR field. After expansion, CR logical instructions take slots 0 and 1. The following three cases summarize the expansion of the CR logical instructions (where E is an eCR):

C = A op B expands to E = A op B; C = E

A = A op B expands to E = A op B; A = E

B = A op B expands to B = A op B; No-op

At the dispatch stage, each of these CR fields is mapped to one of the 32 physical CR fields in the CR Register file. If a conditional branch instruction is dependent on the result of a non-destructive CR logical operation, the earliest it can be executed is four cycles after the first operation of the CR logical instruction is executed.

6.3.11.2 Synchronization Instructions

sync, lwsync, ptesync

The **sync** instruction is a single instruction in its own group. When reaching dispatch, the **sync** is held until all previous loads are done (no loads are left in issue queues and there are no valid LMQ entries) and all previous stores have issued (no stores are left in the issue queues). When these actions have completed, the **sync** and following instruction can be dispatched. To the LSU, a **sync** looks like a st-AGEN instruction, but without an address. The **sync** enters the SRQ and, when it is the next to complete, it is sent to the STS. The **sync** remains in the SRQ until the STS completes its operations, which can take a large number of cycles. Younger loads (I equals '0') are allowed to execute speculatively. However, when the **sync** completes, if any of these loads in the LRQ have had their cache line invalidated on a snoop, then all instructions after the **sync** are flushed. See also *Synchronization Operations* on page 103.

From the core point of view, a **ptesync** is identical to a **sync**, except a **ptesync** sets the CSI-required bit when it finishes (see *isync* for more information).

The **lwsync** instruction is held after reaching dispatch for the same conditions as **sync**. Once these conditions have been met, **lwsync** dispatches and is entered in the SRQ. The **lwsync** instruction is sent to the STS, but no response is returned so the instruction completes and is removed from the SRQ like other stores.

eieio

The **eieio** instruction is different from **sync** in that it does not stall dispatch in the processor. When a younger load, where I equals '1', executes, it will be rejected due to the **eieio** instruction in the SRQ. When the **eieio** is the next to complete and the oldest entry in the SRQ, it is sent to the STS. After that, it is removed from the SRQ. See also *Synchronization Operations* on page 103.

isync

The **isync** instruction is held after reaching dispatch for the same conditions as **sync**. The hardware maintains a CSI-required bit to indicate when the **isync** needs to perform a context synchronizing flush. Once it is next to complete, if the CSI-required bit is set, then **isync** causes all following instructions to be flushed. After the flush, or once it is the next to complete if no flush is required, **isync** completes.

Instructions that set the CSI-required bit include: **icbi**, **mtmsr**, **mtmsrd**, **mtsr**, Move to Segment Register Indirect (**mtsrin**), SLB Move to Entry (**slbmte**), SLB Invalidate Entry (**sibie**), SLB Invalidate All (**slbia**), TLB Invalidate All (**tlbia**), **ptesync**, and **mtspr** instructions that go to FXU or LSU SPRs. In addition, an **icbi** or **ptesync** snooped from another processor sets the CSI-required bit. Instructions that clear the CSI-required bit include: **sc**, **rfid**, **isync**, and interrupts. In addition, flushes can clear the CSI-required bit if they flush the next-to-complete group.

Note: If an **slbmte** instruction alters the mapping, or associated attributes, of a currently mapped effective segment ID (ESID), the **slbmte** must be preceded by an **slbie** (or **slbia**) instruction that invalidates the existing translation. This applies even if the corresponding entry is no longer in the SLB. No software synchronization is needed between the **slbie** and the **slbmte** regardless of whether the index of the SLB entry (if any) containing the current translation is the same as the SLB index specified by the **slbmte**.

No **slbie** or **slbia** is needed if the **slbmte** instruction replaces a valid SLB entry with a mapping of a different ESID (for example, to satisfy an SLB miss). However, the **slbie** is needed later if and when the translation that was contained in the replaced SLB entry is to be invalidated.

6.3.11.3 Cache and Synchronization Operations in STS

See *Section 3.5.3 Non-Cacheable Unit* on page 98 for more information.

6.3.11.4 System Linkage Instructions

sc, rfid

Each of these instructions is in a single instruction group. These instructions are executed entirely by the completion unit. When the instruction is the next to complete and the new address is available, all groups after (subsequent instructions) are flushed. Instructions are refetched from the new address.

6.3.11.5 Stalls in Floating-Point Instruction Processing

Due to the conditions listed below, which are based on the data value (operands or results) and data dependency, one or both of the FPUs may not be available to issue any dependent floating-point operations:

1. ***Back-to-Back Dependency***: If two floating-point operations have a back-to-back dependency, there must be at least six cycles after the first operation is issued before the dependent operation can be issued to the FPU. However, the ISU can issue other independent floating-point operations in the interim.
2. ***Denormalized Stall***: If the input to a floating-point arithmetic operation is not normalized, it needs to be normalized. If only one input needs to be normalized, then the FPU is stalled for eight cycles. The FPU is stalled for 11 cycles, when two inputs need to be normalized or 14 cycles when three inputs need to be normalized.
3. ***Massive Cancellation Stall***: If an arithmetic operation involves an effective subtraction and the two operands of the subtraction operation are very close in magnitude, then the FPU is stalled for two cycles.

More specifically, for the operation A × C + B, if the result is 2^{-16} or less times AxC, then there is a two cycle stall. If the result is 2^{-80} or less times AxC, then there is a four cycle stall. These stalls may sometimes occur for results that are up to 8 times larger (that is 2^{-13} times AxC).

4. ***Underflow/Overflow Stall***: If the exponent of the result of an arithmetic operation is not within the maximum and minimum exponents allowed by the architecture, then there is a two cycle stall. An underflow or overflow stall may follow a massive cancellation. Thus, in the worst case, the result generation may cause a 6-cycle stall for the FPU. It is not required that the answer actually overflow or underflow for this stall to occur. It may also occur for answers close to the overflow or underflow limit.
5. ***Convert to Integer Stall***: If the integer value of the floating-point number exceeds the maximum value that can be stored in the integer word (or double word), then there is a 2-cycle stall.
6. ***Convert from Integer Stall***: Any such conversion has a 2-cycle stall.
7. ***Busy for Divide and Square-Root Operation***: A divide takes a minimum of 28 cycles and a square-root takes a minimum of 35 cycles. It takes two to three cycles to detect special cases (such as divide by zero or the square root of a negative number) and stop the operation. Therefore, for these two to three cycles, the FPU signals busy to the ISU. Division and square-root determination is done using an iterative process. The initial approximate value of the final result is obtained using a 128-entry table for division or a 32-entry table for a square-root determination. Because the initial value is close to the final result, there is a higher probability that the iterative process will generate a massive cancellation stall. There could be at most two massive cancellation stalls during a divide or square-root operation. There might be additional denormalized stalls if the input operands are not normalized, and an underflow or overflow stall if the result is not in the normal range. A dependent operation must wait an additional five cycles before being issued.

Back-to-back dependency is detected and managed by the ISU. The busy signal due to divide and square-root operations is initiated by the ISU and terminated by the FPU. During this period, the ISU does not issue instructions to the FPU that is busy (while it can issue instructions to the other FPU). All other stall signals are generated by the FPU. During these stalls, the ISU does not issue instructions to either of the two FPUs, and also waits one additional cycle before issuing to them. Thus, a 2-cycle stall at the FPU is a 3-cycle stall at the ISU, and so on.

6.3.11.6 Delays in Instruction Processing

Table 6-13 shows the issue-to-issue delays for dependent operations involving instructions.

Table 6-13. Issue-to-Issue Delays for Dependent Operations

From	to LSU	to FXU	to FPU	to CRU	to VPERM	to VPU(ALU)
LSU	3	3	5	3	4	5
FXU	—	2	—	3	—	—
FPU	—	—	6	8	—	—
CRU	—	—	—	2	—	—
VPU perm	—	—	—	—	2	3
VPU simple	—	—	—	—	3	2
VPU complex	—	—	—	—	6	5
VPU float	—	—	—	—	9	8

6.4 Key Latencies, Throughputs, and Bandwidths

This section provides information about many of the key latencies, throughput rates, and bandwidths of the 970MP processing unit. Throughout this section, it is assumed that the reader is familiar with the PowerPC Architecture and the 970MP processing unit pipeline.

6.4.1 Instruction Latencies and Throughputs

Instruction-level performance in an out-of-order, multi-issue machine like the 970MP microprocessor is difficult to describe and may be somewhat misleading. Many of the traditional sensitivities and resulting effects can actually be absorbed by the various queues and by the dynamic instruction scheduling implicit in the machine. As a result, while it is still important to understand instruction-level characteristics, it is also important to take a much broader look at the machine to really understand the way things perform.

Table 6-14 through *Table 6-19* summarize the raw instruction-level performance characteristics of the 970MP processing unit. Some comments on the tables are relevant:

- During the decode stages, the 970MP processing unit can crack instructions, emulate instructions with microcode, or let them flow through unobstructed.
- Microcoded instructions are forced into a group by themselves. From there, the microcode itself may form one or more groups to emulate the original instruction.
- Some instructions are executed in one or more pipelines. The following tables note to which pipelines an instruction is dispatched. In addition, if an instruction is dispatched to the FXU, load/store, or FPU pipeline, it normally can be executed in either of the two pipelines, unless noted otherwise in the following tables.
- Latency is the issue-to-issue latency for a dependent instruction. For example, if an **add** issues in cycle 20 and the soonest a dependent **xor** could issue is cycle 22, the latency for **add** is listed as 2. There are multiple latencies for some instructions when the instruction writes more than one type of register.
- Throughput refers to the maximum sustained rate the 970MP processing unit can execute instructions of the type noted in the absence of any dependencies assuming infinite caches. It is shown as instructions per cycle (IPC).
- Some fields are labelled "N/A" to indicate that a more elaborate description is required. This includes complex microcoded instructions. Instructions that do not modify a register are also labelled "N/A."
- The **sync**, **lwsync**, **eieio** instructions:
  - Formed into single-instruction group by the instruction dispatch unit (IDU).
  - Held in dispatch until there are no LSU instructions in issue queues, and no entries used in the LMQ (that is, all loads have come home).
  - Dispatched to issue queue. This creates an entry in the store queue (STQ).
  - At this time, following instructions have no special restrictions imposed by the instruction.
  - Issued and executed in LSU (LSU0 because of position as first operation in a group).
  - When an STQ entry is the next to process (that is, all earlier stores have drained), the **sync** operation is sent to the L2. This completes **eieio** and **lwsync** instructions.
  - **sync** completes when the transaction is acknowledged by the STS (after going all the way out to the bus).

Tables 6-14 through *6-19* list latencies associated with instructions executed in the designated pipeline.

6.4.1.1 Branch Instruction Characteristics

Table 6-14 lists latencies for the instructions executed in the branch pipeline.

Table 6-14. Branch Instruction Characteristics

Instruction	Execute			Other interlocks	Other Comments
	Pipeline	Latency	Throughput (IPC)		
b ba bl bla	Branch	N/A	1/cycle	No	Unconditional branches are always predicted correctly.
bc bca bcl bcla	Branch	2 cycles (LR/CTR)	1/cycle	No	Branch direction predicted at top of pipeline; may update the link stack.
bclr bclrl	Branch	2 cycles (LR/CTR)	1/cycle	No	Branch direction and target address predicted at top of pipeline; may make use of link stack, or count cache, or both.
bcctr bcctrl	Branch	2 cycles (LR/CTR)	1/cycle	No	Branch direction and target address predicted at top of pipeline; may make use of link stack, or count cache, or both.
mtspr(LR) **mtspr**(CTR)	Branch	3 cycles	1/cycle	No	
mtspr(others) **mtmsr mtmsrd**	Either FXU or branch (depends on SPR)	Varies based on SPR	Varies based on SPR	Wait for non-rename scoreboard bit to clear.	
mfspr(LR) **mfspr**(CTR)	Branch	3 cycles	1/cycle	No	

LR: Link Register; CTR: Count Register; SPR: Special Purpose Register

6.4.1.2 Instruction Characteristics (Condition Register Logic)

Table 6-15 lists latencies for the instructions executed in the Condition Register logic pipeline.

Table 6-15. Instruction Characteristics (Condition Register Logic)

Instruction	Execute			Other interlocks	Other Comments
	Pipeline	Latency	Throughput (IPC)		
crand cror crxor crnand crnor crnandc creqv crorc (non-destructive)	CRlogic	4 cycles	1/cycle	No	CR logicals are non-destructive if BT and BB are not in the same CR field.
crand cror crxor crnand crnor crnandc creqv crorc (destructive)	CRlogic	2 cycles	1/cycle	No	CR logicals are destructive if BT and BB are in the same CR field.
mcrf	CRlogic	2 cycles	1/cycle	No	
mfcr	CRlogic	N/A	N/A	No	
mfcrf	CRlogic	3 cycles	1/cycle	No	

6.4.1.3 Instruction Characteristics (Load/Store)

Table 6-16 lists latencies for the instructions executed in the load/store pipeline.

The 970MP core passes stores to the STS at the rate of one store per cycle. The storage subsystem can further reduce the store bandwidth. However, up to two stores/cycle are completed on the L2/STS. The reduced store to STS bandwidth slows the processor down only when the STS causes the core's store queue to fill, thereby blocking additional stores from dispatching.

Latency for fixed-point instructions that set the Condition Register is shown from the fixed-point instruction to a conditional branch instruction. The latency to a CR logical instruction is one additional cycle.

Table 6-16. Instruction Characteristics (Load/Store) (Page 1 of 4)

Instruction	Execute			Other interlocks	Other Comments
	Pipeline	Latency	Throughput (IPC)		
dcbf	ld/st	N/A	1/cycle		
dcbst	ld/st	N/A	1/cycle		
dcbt dbtst	ld/st	N/A	1/cycle		
dcbz	ld/st	N/A	1/cycle		Invalidates the L1 cache line on its way to the L2. Allocation and zero function occur at the L2 cache.
eieio	ld/st	N/A	N/A	Wait until it is next-to-complete and for all prior load data to come home.	See *Table 6-21 Storage Alignment Characteristics* on page 215.
icbi	ld/st	N/A	N/A		After load/store generates and translates the EA, the **icbi** looks like a snooped **icbi** to the instruction fetcher. The snooped **icbi** spins through 16 possible locations for the block (I-cache is indexed with effective address, so aliasing can occur).
isync	ld/st	N/A	N/A	Wait until it is next-to-complete and for all prior load data to come home.	Causes a flush and instruction refetch only if it is preceded by instructions that change the content of the machine or **icbi** or **ptesync**.
lbz lbzx lhz lhzx lwz lwzx ld ldx lhbrx lwbrx	ld/st	3 cycles	2/cycle	No	See *Table 6-21 Storage Alignment Characteristics* on page 215 and *Section 6.4.2 Storage Alignment Performance Characteristics* on page 215.
lbzu lhzu lwzu ldu	ld/st, fxu	3 cycles (**r**D) 2 cycles (**r**A)	2/cycle	No	Cracked into a basic load and an add.
lbzux lhzux lwzux ldux	ld/st, fxu	3 cycles (**r**D) 4 cycles (**r**A)	1/cycle	N/A	Broken into a basic load and an add, but must avoid disturbing **r**B, which adds another IOP.
lfs lfsx lfd lfdx	ld/st	5 cycles	2/cycle	No	Instructions are executed in the LSU (the FPU just arranges rename). See *Table 6-21 Storage Alignment Characteristics* on page 215.

rD: Field used to specify a General Purpose Register (GPR) to be used as a target.
rA: Field used to specify a GPR to be used as a source or as a target.
rB: Field used to specify a GPR to be used as a source.

Table 6-16. Instruction Characteristics (Load/Store) (Page 2 of 4)

Instruction	Execute			Other interlocks	Other Comments
	Pipeline	Latency	Throughput (IPC)		
lfsu lfsux lfdu lfdux	ld/st, fxu	5 cycles (FPR) 2 cycles (GPR)	2/cycle	No	Cracked into normal **load** and an FXU **add**. See *Table 6-21 Storage Alignment Characteristics* on page 215.
lha lhax lwa lwax	ld/st, FXU	5 cycles	2/cycle	No	Cracked into a basic **load** and an Extend Sign (**exts**).
lhau	ld/st, FXU	5 cycles (**r**D) 2 cycles (rA)	1/cycle	N/A	Broken into a basic **load**, an **exts**, and an **add.**
lhaux lxaux	ld/st, FXU	5 cycles (**r**D) 4 cycles (rA)	2/3cycles	N/A	
lmw	ld/st	N/A	2 register loads per cycle after start-up	N/A	Number of dispatch groups is equal to (number of registers modulo four). Microcode generates an inline sequence of basic loads.
lswi lswx stswi stswx (unaligned)	ld/st	N/A	N/A	N/A	String instruction is first decoded and dispatched as described above. At execute, the ld/st notes that it is unaligned and causes a machine flush. As the string instruction goes through decode the second time, it is broken up in a way that takes the misalignment into account.
lswi (naturally aligned)	ld/st	N/A	2 register loads per cycle after start-up	N/A	Number of dispatch groups is approximately equal to (number of registers modulo four). Microcode assumes natural alignment and generates inline sequence of basic loads.
lswx (naturally aligned)	ld/st	N/A	2 register loads per cycle after start-up	N/A	Number of dispatch groups is approximately equal to (number of registers modulo four) plus 4 more for startup. Microcode assumes natural alignment and generates inline sequence of basic loads.
lwarx ldarx	ld/st	N/A	N/A	No	Forced to miss data L1 cache.
lwsync	ld/st	N/A	N/A	Wait until it is next-to-complete and for all prior load data to come home.	Forces previous stores to finish into the cache/memory hierarchy (that is, out of the STQs). Still broadcasts **sync** transaction onto bus (but does not block).
mfsr mfsrin	ld/st	3 cycles	1/cycle	No	
mtsr mtsrin	ld/st			Wait for non-rename scoreboard bit to clear.	Causes selective invalidates out of the I-ERAT and D-ERAT.
ptesync	ld/st	N/A	N/A	Wait until it is next-to-complete and for all prior load data to come home.	Forces previous stores to finish into the cache/memory hierarchy (that is, out of the STQs).
slbia	ld/st	N/A			Fully invalidates the SLB, the I-ERAT, and the D-ERAT.
slbie	ld/st	N/A			Causes index-based invalidate in both the I-ERAT and the D-ERAT.

rD: Field used to specify a General Purpose Register (GPR) to be used as a target.
rA: Field used to specify a GPR to be used as a source or as a target.
rB: Field used to specify a GPR to be used as a source.

Table 6-16. Instruction Characteristics (Load/Store) (Page 3 of 4)

Instruction	Execute			Other interlocks	Other Comments
	Pipeline	Latency	Throughput (IPC)		
slbmfev slbmfee	ld/st	N/A		Wait for the non-rename scoreboard bit to clear.	
slbmte	ld/st	N/A		wait for non-rename scoreboard bit to clear.	Causes selective invalidates out of the I-ERAT and D-ERAT.
stb sth stw std	ld/st, FXU	N/A	2/cycle	No	Stores are internally issued twice (once as an ea_gen operation, and once as a data_steer operation). Stores put their data into the STQ, which later writes to cache/memory hierarchy (the STQ supports forwarding for many cases).
stbu sthu stwu stdu	ld/st, FXU	2 cycles for updated register	1/cycle	No	Cracked into basic **store** and an **add**.
stbux sthux stwux stdux	ld/st, FXU	2 cycles for updated register	1/cycle	N/A	Broken into three IOPs.
stbx sthx stwx stdx	ld/st, FXU	N/A	2/cycle	No	Three source operands are cracked to minimize the breadth of the renaming facility.
stfiwx	ld/st, FPU	N/A	2/cycle	No	
stfs stfsx stfd stfdx	ld/st, FPU	N/A	2/cycle	No	Instructions are dispatched to both the FPU and the LSUs. See *Table 6-21 Storage Alignment Characteristics* on page 215.
stfsu stfsux stfdu stfdux	ld/st, FPU, FXU	2 cycles for GPR	2/cycle	No	Cracked into a normal **store** and an FXU **add**. Normal **store** instructions are dispatched to both the FPU and the LSUs. See *Table 6-21 Storage Alignment Characteristics* on page 215.
sthbrx stwbrx	ld/st, FXU	N/A	2/cycle	No	Cracked into basic **store** and **byte manipulating** IOP.
stmw	ld/st, FXU	N/A	2 register stores per cycle after start-up	N/A	Number of dispatch groups is equal to (number of registers modulo four). Microcode generates an inline sequence of basic stores.
stswi (naturally aligned)	ld/st, FXU	N/A	2 register stores per cycle after start-up	N/A	Number of dispatch groups is approximately equal to (number of registers modulo four). Microcode assumes natural alignment and generates inline sequence of basic stores.
stswx (naturally aligned)	ld/st, FXU	N/A	2 register stores per cycle after start-up	N/A	Number of dispatch groups is approximately equal to (number of registers modulo four) plus four more for startup. Microcode assumes natural alignment and generates an inline sequence of basic stores.

rD: Field used to specify a General Purpose Register (GPR) to be used as a target.
rA: Field used to specify a GPR to be used as a source or as a target.
rB: Field used to specify a GPR to be used as a source.

Table 6-16. Instruction Characteristics (Load/Store) (Page 4 of 4)

Instruction	Execute			Other interlocks	Other Comments
	Pipeline	Latency	Throughput (IPC)		
stwcx. stdcx.	ld/st, FXU	N/A	N/A	No	Must establish coherency block ownership before completing the instruction (other stores do not have to do this). This can take anywhere from 10 cycles to hundreds of cycles, depending upon the state of the coherency block in the memory hierarchy.
sync	ld/st	N/A	N/A	Wait until it is next-to-complete and for all prior load data to come home.	Forces previous stores to finish into the cache/memory hierarchy (that is, out of the STQs).
tlbie	ld/st	N/A			Causes index-based invalidate in both the I-ERAT and the D-ERAT. Gets broadcast onto the bus.
tlbiel	ld/st	N/A			Causes index-based invalidate in both the I-ERAT and the D-ERAT. Does not get broadcast onto the bus.
tlbsync	ld/st	N/A			

rD: Field used to specify a General Purpose Register (GPR) to be used as a target.
rA: Field used to specify a GPR to be used as a source or as a target.
rB: Field used to specify a GPR to be used as a source.

6.4.1.4 Instruction Characteristics (FXU)

Table 6-17 lists latencies for the instructions executed in the FXU pipeline. The latency for fixed-point instructions that set the Condition Register is shown from the fixed-point instruction to a conditional branch instruction. The latency to a CR logical instruction is one additional cycle.

Table 6-17. Instruction Characteristics (FXU) (Page 1 of 4)

Instruction	Execute			Other interlocks	Other Comments
	Pipeline	Latency	Throughput (IPC)		
addc subfc	FXU	2 cycles (GPR)	1/cycle	Wait for non-rename scoreboard bit to clear.	
addc. addco addco. subfc. subfco subfco.	FXU	2 cycles (GPR) 5 cycles (CR)	2/3 cycles	Wait for non-rename scoreboard bit to clear.	
addeo. addmeo. subfmeo. addzeo. subfzeo.	FXU	2 cycles (GPR) 5 cycles (CR)	2/3 cycles	No	These operations may architecturally change the summary overflow bit. The first attempted execution of these assumes that the SO bit will not change. If it does, the instruction will cause a flush and then be re-executed.
addi addis add add. subf subf. addic subfic adde addme subfme addze. subfze neg neg. mego	FXU	2 cycles (GPR) 3 cycles (CR)	2/cycle	No	

Table 6-17. Instruction Characteristics (FXU) (Page 2 of 4)

Instruction	Execute			Other interlocks	Other Comments
	Pipeline	Latency	Throughput (IPC)		
addic. adde. subfe. addme. subfme. addze. subfze.	FXU	2 cycles (GPR) 5 cycles (CR)	1/cycle	No	Cracked into a basic **add (sub)** and a Compare (**cmp**).
addo subfo	FXU	2 cycles	2/cycle	No	These operations can architecturally change the summary overflow bit. The first attempted execution of these assumes that the SO bit will not change. If it does, the instruction will cause a flush and then be re-executed.
addo. subfo. addeo subfeo addmeo subfmeo addzeo subfzeo nego.	FXU	2 cycles (GPR) 5 cycles (CR)	1/cycle	No	These operations can architecturally change the summary overflow bit. The first attempted execution of these assumes that the SO bit will not change. If it does, the instruction will cause a flush and then be re-executed.
andi. andis. ori oris xori xoris and and. or or. xor xor. nand nand. nor nor. eqv eqv. andc andc. orc orc.	FXU	2 cycles (GPR) 3 cycles (CR)	2/cycle	No	
cmpi cmp cmpli cmpl	FXU	3 cycles	2/cycle	No	
cntlzd cntlzd. cntlzw cntlzw.	FXU	2 cycles (GPR) 5 cycles (CR)	2/cycle	No	
divd divdu	FXU (one)	68 cycles	1/67 cycles	No	Not pipelined in the FXU. Cracked into a no-op and a **divd** (**divdu**). This forces the divide into the issue queue associated with the fixed-point unit capable of doing divides.
divd. divw. divdu. divdw.	FXU (one)	Same as above for GPR + 3 cycles for CR	Same as above	No	Not pipelined in the FXU. Broken into three IOPs: a no-op, a divide, and a **cmp**.
divdo divwo divduo divwuo	FXU (one)	Same as above for GPR	Same as above	No	Not pipelined in the FXU. Cracked into a no-op and a Divide Doubleword (Divide Doubleword Unsigned) **divd** (**divdu**). This forces the divide into the issue queue associated with the fixed-point unit capable of doing divides. These operations can architecturally change the summary overflow bit. The first attempted execution of these assumes that the SO bit will not change. If it does, the instruction will cause a flush and then be re-executed.
divdo. divwo. divduo. divwuo.	FXU (one)	Same as above for GPR + 3 cycles for CR	Same as above	No	Not pipelined in the FXU. Broken into three IOPs: a no-op, a divide and a **cmp**. These operations can architecturally change the summary overflow bit. The first attempted execution of these assumes that the SO bit will not change. If it does, the instruction will cause a flush and then be re-executed.
divw divwu	FXU (one)	36 cycles	1/35 cycles	No	Cracked into a no-op and a **divd** (**divdu**). This forces the divide into the issue queue associated with the fixed-point unit capable of doing divides.

Table 6-17. Instruction Characteristics (FXU) (Page 3 of 4)

Instruction	Execute			Other interlocks	Other Comments
	Pipeline	Latency	Throughput (IPC)		
extsb extsb. extsh extsh. extsw extsw.	FXU	2 cycles (GPR) 5 cycles (CR)	2/cycle	No	
mfspr(others) **mfmsr**	FXU (one)	Varies based on SPR	Varies based on SPR	Wait for non-rename scoreboard bit to clear.	Use of the FXU pipeline is blocked while waiting for the SPR value.
mfspr(xer)	FXU (one)	8 cycles	N/A	Wait for non-rename scoreboard bit to clear.	Use of the FXU pipeline is blocked while waiting for the SPR value.
mftb	FXU (one)	~10 cycles	1/~10 cycles	Wait for non-rename scoreboard bit to clear.	Use of the FXU pipeline is blocked while waiting for the SPR value.
mtcrf (one field) **mtocrf**	FXU (one)	N/A	1/cycle	No	
mtcrf (more than one field)	FXU (one)	N/A	N/A	No	
mtspr(others) **mtmsr mtmsrd**	Either FXU or branch (depends on SPR)	Varies based on SPR	Varies based on SPR	Wait for non-rename scoreboard bit to clear.	
mtspr(xer)	FXU (one)	N/A	N/A	Wait for non-rename scoreboard bit to clear.	SPR movement instructions can only be done by one of the FXU pipelines (the other one does divides).
mulld mulhd mulhdu	FXU	7 cycles	2/6 cycles	No	Not pipelined in the FXU.
mulli	FXU	4 cycles	2/3 cycles	no	Not pipelined in the FXU.
mullw mulhw mulhwu	FXU	5 cycles	2/4 cycles	no	Not pipelined in the FXU.
mullw. mulld. mulhd. mulhw. mulhdu. mulhwu.	FXU	Same as above for GPR + 3 cycles for (CR)	2/5 (7) cycles	No	Not pipelined in the FXU. Cracked into a baseline operation and a **cmp**.
mullwo mulldo	FXU	5 (7) cycles for **mullwo** (**mullwdo**)	2/4 (6) cycles for mullwo (mulldo)	No	Not pipelined in the FXU. These operations can architecturally change the summary overflow bit. The first attempted execution of these assumes that the SO bit will not change. If it does, the instruction will cause a flush and then be re-executed.
mullwo. mulldo.	FXU	Same as above for GPR + 3 cycles for (CR)	2/5 (7) cycles	No	Not pipelined in the FXU. Cracked into a baseline operation and a **cmp**. These operations can architecturally change the summary overflow bit. The first attempted execution of these assumes that the SO bit will not change. If it does, the instruction will cause a flush and then be re-executed.

Table 6-17. Instruction Characteristics (FXU) (Page 4 of 4)

Instruction	Execute			Other interlocks	Other Comments
	Pipeline	Latency	Throughput (IPC)		
rldicl rldicl. rldicr rldirc. rldic rldic. rlwinm rlwinm. rldcl rldcl. rldcr rldcr. rlwnm rlwnm. rldimi rldimi.	FXU	2 cycles (GPR) 5 cycles (CR)	2/cycle 1/cycle when RC equals '1'	No	
rlwimi	FXU	4 cycles (GPR)	1/cycle	No	
rlwimi.	FXU	4 cycles (GPR) 7 cycles (CR)	2/3 cycles	No	
sld sld. slw slw. srd srd. srw srw. sradi srawi srad sraw	FXU	2 cycles (GPR) 5 cycles (CR)	2/cycle 1/cycle when RC equals '1'	No	
sradi. srawi. srad. sraw.	FXU	2 cycles (GPR) 5 cycles (CR)	1/cycle	No	
tdi twi td tw	FXU	N/A	2/cycle	No	

6.4.1.5 Instruction Characteristics (FPU)

Table 6-18 lists latencies for the instructions executed in the FPU pipeline.

Table 6-18. Instruction Characteristics (FPU) (Page 1 of 2)

Instruction	Execute			Other interlocks	Other Comments
	Pipeline	Latency	Throughput (IPC)		
fcmpu fcmpo	FPU	8 cycles	2/cycle	No	
fctid fctidz fctiw fctiwz fcfid	FPU	6 cycles	2/cycle	No	
fdiv fdivs	FPU	33 cycles	2/28 cycles	No	Makes use of microcoded sequence within the floating-point unit. Additional cycles may be needed for stalls during microcode execution. See *Section 6.3.11.5 Stalls in Floating-Point Instruction Processing* on page 201.
fdiv. fdivs. **fsqrt. fsqrts.** **fres. frsqrte.**	FPU	Same as above + 2 cycles for CR	1/28 cycle 1/35 cycle 1/cycle	No	Other non-dotted commands are allowed to overlap execution in the other FPU.
fmadd fmadds fmsub fmsubs fnmadd fnmadds fnmsub snmsubs	FPU	6 cycles	2/cycle	No	
fmadd. fmadds. fmsub. fmsubs. fnmadd. fnmadds. fnmsub. fnmsubs.	FPU	6 cycles (FPR) 8 cycles (CR)	1/cycle	No	

Table 6-18. Instruction Characteristics (FPU) (Page 2 of 2)

Instruction	Execute			Other interlocks	Other Comments
	Pipeline	Latency	Throughput (IPC)		
fmr fneg fabs fnams fadd fadds fsub fsubs fmul fmuls	FPU	6 cycles	2/cycle	No	
fmr. fneg. fabs. fnams. fadd. fadds. fsub. fsubs. fmul. fmuls.	FPU	6 cycles (FPR) 8 cycles (CR)	1/cycle	No	
fres frsqrte	FPU	6 cycles	2/cycle	No	Table look-up method.
frsp	FPU	6 cycles	2/cycle	No	
frsp. fctid. fctidz. fctiw. fctiwz. fcfid.	FPU	Same as above + 2 cycles for CR	1/cycle	No	
fsel	FPU	6 cycles	2/cycle	No	
fsel.	FPU	Same as above + 2 cycles for CR	1/cycle	No	
fsqrt fsqrts	FPU	40 cycles	2/35 cycles	No	
mcrfs	FPU		1/cycle	No	
mffs mffs.	FPU		1/cycle	No	
mtfsfi mtfsfi. mtfsf mtfsf. mtfsb0 mtfsb0. mtfcb1 mtfsb1.	FPU		1/cycle	No	See *Table 6-9 Decode Slot/Dispatch Restrictions* on page 179 when RC equals '1'.

6.4.1.6 Instruction Characteristics (Miscellaneous)

Table 6-19 lists latencies for some instructions not executed in an execution unit.

Table 6-19. Instruction Characteristics (Miscellaneous)

Instruction	Execute			Other interlocks	Other Comments
	Pipeline	Latency	Throughput (IPC)		
sc rfid	ISU	N/A	N/A	No	
ori 0,0,0 (preferred no-op)	None	Zero cycles	4/cycle	No	A special form of the OR Immediate (**ori**) instruction is finished at the same time that it is dispatched.

6.4.1.7 Instruction Characteristics (Vector)

For additional information about the vector instruction set, see *Section 13.6 Vector Instruction Set* on page 494.

Table 6-20. Instruction Characteristics (Vector) (Page 1 of 3)

Instruction	Execute			Other Comments
	Pipeline	Latency	Throughput (IPC)	
dss dssall	ld/st/FXU	N/A	1/cycle	Wait until it is the next to complete.
dst dstt	ld/st/FXU	N/A	1/cycle	Wait until it is the next to complete.
dstst dststt	ld/st/FXU	N/A	1/cycle	Wait until it is the next to complete.
mtvrsave mfvrsave	FXU	2 cycles (GPR) 3 cycles (CR)	2/cycle	Same as an **or**.
mtvscr mfvscr	VALU	N/A	N/A	Forced to be the only instruction in a group.
lvebx lvehx lvewx lvx lvxl	ld/st	4/5 cycles	1/cycle	Latency is four cycles for a VPERM operation, five cycles for a VALU operation.
stvebx stvehx stvewx-vstvx stvxl	ld/st, VALU	N/A	1/cycle	
lvsl lvsr	ld/st	4/5 cycles	1/cycle	Latency is four cycles for a VPERM operation, five cycles for a VALU operation.
vaddubm vaddubs vaddsbs vadduhm vadduhs vaddshs vadduwm vadduws vaddsws	VALU	2 cycles	1/cycle	
vaddfp	VALU	8 cycles	1/cycle	
vaddcuw	VALU	2 cycles	1/cycle	
vsububm vsububs vsubsbs vsubuhm vsubuhs vsubshs vsubuwm vsubuws vsubsws vsubfp	VALU	2 cycles	1/cycle	
vsubcuw	VALU	2 cycles	1/cycle	
vmuloub vmulosb vmulouh vmulosh	VALU	5 cycles	1/cycle	
vmuleub vmulesb vmuleuh vmulesh	VALU	5 cycles	1/cycle	
vmhaddshs vmhraddshs vmladduhm vmaddfp	VALU	5 cycles	1/cycle	
vmsumubm vmsummbm vmsumuhm vmsumuhs vmsumshm vmsumshs	VALU	5 cycles	1/cycle	
vsumsws	VALU	5 cycles	1/cycle	
vsum2sws	VALU	5 cycles	1/cycle	

Table 6-20. Instruction Characteristics (Vector) (Page 2 of 3)

Instruction	Execute			Other Comments
	Pipeline	Latency	Throughput (IPC)	
vsum4ubs vsum4sbs vsum4shs	VALU	5 cycles	1/cycle	
vavgub vavgsb vavguh vavgsh vavguw vavgsw	VALU	2 cycles	1/cycle	
vand vor vxor vandc vnor	VALU	2 cycles	1/cycle	
vrlb vrlh vrlw	VALU	2 cycles	1/cycle	
vslb vslh vslw vsl	VALU	2 cycles	1/cycle	
vsrb vsrab vsrh vsrah vsrw vsraw vsr	VALU	2 cycles	1/cycle	
vcmpgtub[.] **vcmpgtsb**[.] **vcmpgtuh**[.] **vcmpgtsh**[.] **vcmpgtuw**[.] **vcmpgtsw**[.] **vcmpgtfp**[.]	VALU	2 cycles	1/cycle	
vcmpequb[.] **vcmpequh**[.] **vcmpequw**[.] **vcmpeqfp**[.]	VALU	2 cycles	1/cycle	
vcmpgefp[.]	VALU	2 cycles	1/cycle	
vcmpbfp[.]	VALU	2 cycles	1/cycle	
vsel	VALU	2 cycles	1/cycle	
vpkuhum vpkuhus vpkshus vpkshss vpkuwum vpkuwus vpkswus vpkswss vpkpx	VPERM	2 cycles	1/cycle	
vupkhsb vupkhsh vupkhpx	VPERM	2 cycles	1/cycle	
vupklsb vupklsh vupklpx	VPERM	2 cycles	1/cycle	
vmrghb vmrghh vmrghw	VPERM	2 cycles	1/cycle	
vmrglb vmrglh vmrglw	VPERM	2 cycles	1/cycle	
vspltb vsplth vspltw	VPERM	2 cycles	1/cycle	
vspltisb vspltish vspltisw	VPERM	2 cycles	1/cycle	
vperm	VPERM	2 cycles	1/cycle	
vsldoi	VPERM	2 cycles	1/cycle	
vslo vsro	VPERM	2 cycles	1/cycle	

Table 6-20. Instruction Characteristics (Vector) (Page 3 of 3)

Instruction	Execute			Other Comments
	Pipeline	Latency	Throughput (IPC)	
vmaxub vmaxsb vmaxuh vmaxsh vmaxuw vmaxsw vmaxfp	VALU	2 cycles	1/cycle	
vminub vminsb vminuh vminsh vminuw vminsw vminfp	VALU	2 cycles	1/cycle	
vrefp vrsqrtefp vlogefp vexptefp	VALU	8 cycles	1/cycle	
vnmsubfp	VALU	8 cycles	1/cycle	
vrfin vrfiz vrfip vrfim	VALU	8 cycles	1/cycle	
vctuxs vctsxs	VALU	8 cycles	1/cycle	
vcfux vcfsx	VALU	8 cycles	1/cycle	

6.4.2 Storage Alignment Performance Characteristics

The following table summarizes 970MP processing unit's performance on various storage alignment cases:

Table 6-21. Storage Alignment Characteristics (Page 1 of 2)

Operand			Alignment					
Type	Size (Bytes)	Byte Alignment	Within 8-Byte Block	Cross 8-Byte Boundary	Cross 32-Byte Boundary	Cross 64-Byte Boundary	Cross 4-KB Boundary	Cross Segment
Integer load (L1 hit)	1, 2, 4, 8	any	optimal	optimal	optimal	~20 cycles	~20 cycles	~20 cycles
Integer load (L1 miss)	1, 2, 4, 8	any	optimal	optimal	±20 cycles	±20 cycles	±20 cycles	±20 cycles
Integer store[1]	1, 2, 4, 8	any	optimal	optimal	optimal	optimal	~20 cycles	~20 cycles
Floating-point load	4	not word	optimal	optimal	alignment exception	alignment exception	alignment exception	alignment exception
Floating-point load (L1 hit)	4, 8	any	optimal	optimal	optimal	~20 cycles	~20 cycles	~20 cycles
Floating-point load (L1 miss)	4, 8	any	optimal	optimal	±20 cycles	±20 cycles	±20 cycles	±20 cycles
Floating-point store[1]	4, 8	any	optimal	optimal	optimal	optimal	~20 cycles	~20 cycles
Floating-point store[1]	4, 8	not word	optimal	optimal	optimal	optimal	alignment exception	alignment exception

Note:

1. Stores that cross an 8-byte boundary cause two store operations from the 970MP core to the STS, each of which is contained within an 8-byte aligned sector. The additional core-to-STS store does not cause a processor slowdown unless the core's store queues (SRQs) are fully utilized blocking additional stores from being dispatched.

Table 6-21. Storage Alignment Characteristics (Page 2 of 2)

Operand			Alignment					
Type	Size (Bytes)	Byte Alignment	Within 8-Byte Block	Cross 8-Byte Boundary	Cross 32-Byte Boundary	Cross 64-Byte Boundary	Cross 4-KB Boundary	Cross Segment
lmw, **stmw**	any multiple of 4 bytes (word) or 8 bytes (double word)	natural alignment	optimal	optimal	optimal	optimal	optimal	optimal
lmw, **stmw**	any multiple of 4 bytes (word) or 8 bytes (double word)	not natural alignment	poor (alignment exception)	poor (alignment exception)	poor (alignment exception)	poor (alignment exception)	poor (alignment exception)	poor (alignment exception)
load string word, store string word	any	4 bytes	optimal	optimal	optimal	optimal	optimal	optimal
load string word, store string word	any	anything but 4 bytes	optimal	optimal	±20 cycles per instruction (loads) (cache miss case only)	~20 cycles per instruction (loads)	~20 cycles	~20 cycles
cache-inhibited load string word, store string word	any	any	poor (alignment exception)	poor (alignment exception)	poor (alignment exception)	poor (alignment exception)	poor (alignment exception)	poor (alignment exception)
load string double word, store string double word	any	8 bytes	optimal	optimal	optimal	optimal	optimal	optimal
load string double word, store string double word	any	any but 8 bytes	optimal	optimal	+ ~20 cycles per instruction (loads) (cache miss case only)	~20 cycles per instruction (loads)	~20 cycles	~20 cycles
cache-inhibited load string double word, store string double word	any	any	poor (alignment exception)	poor (alignment exception)	poor (alignment exception)	poor (alignment exception)	poor (alignment exception)	poor (alignment exception)
Cache-inhibited load	1, 2, 4, 8	natural alignment	optimal	N/A	N/A	N/A	N/A	N/A
Cache-inhibited load	1, 2, 4, 8	not natural alignment	poor (alignment exception)	poor (alignment exception)	poor (alignment exception)	poor (alignment exception)	poor (alignment exception)	poor (alignment exception)
Cache-inhibited store	1, 2, 4, 8	natural alignment	optimal	N/A	N/A	N/A	N/A	N/A
Cache-inhibited store	1, 2, 4, 8	not natural alignment	poor (alignment exception)	poor (alignment exception)	poor (alignment exception)	poor (alignment exception)	poor (alignment exception)	poor (alignment exception)

Note:

1. Stores that cross an 8-byte boundary cause two store operations from the 970MP core to the STS, each of which is contained within an 8-byte aligned sector. The additional core-to-STS store does not cause a processor slowdown unless the core's store queues (SRQs) are fully utilized blocking additional stores from being dispatched.

6.4.3 Memory System Performance Characteristics

6.4.3.1 Load Latencies

See *Table 3-1 Storage Hierarchy Characteristics* on page 85 for a definition of the storage hierarchy levels. *Table 6-22* lists the key memory latencies (in processor clocks) for processor interconnect bus ratios of 2:1.

Table 6-22. Key Load Latencies

Level	Data Load-Use[1] Bus at 2:1	Instruction Fetch Bus at 2:1
L1 D-cache hit	2	N/A
L1 I-cache hit	N/A	1
L2 hit	14	15

1. Numbers in the table are latencies for FXU and ld/st. Add one cycle for FPU latencies.

6.4.3.2 Peak System Bandwidths

See *Table 3-1 Storage Hierarchy Characteristics* on page 85 for a definition of the storage hierarchy levels. *Table 6-23* lists the key peak bandwidths assuming a processor-to-bus ratio of 2:1.

Table 6-23. Peak System Bandwidths

	Ratio to Processor Clock	Bytes/Cycle	Number	Peak Bandwidth GBps per GHz	Comments
L1 I-cache	1:1	32	1 port	32	
L1 D-cache loads	1:1	8	2 ports	16	L1 D-cache capable of two reads and one write concurrently. Two LSU pipes can execute 8-byte stores concurrently. A write to the L1 D-cache is single-ported.
L1 D-cache stores	1:1	8	1 port	8	
L1 to L2 Stores	1:1	8	1 bus	8	Non VPU store.
L1 to L2 Stores	1:1	16	1 bus	16	VPU store.
L2 to L1 I-cache	1:1	32	1 bus	32	
L2 to L1 D-cache	1:1	32	1 bus	32	
L2 Cache Reads	2:1	64	1 R/W port	32	
L2 Cache Writes	2:1	64	1 R/W port	32	Assumes store gathering to 64 bytes.
Memory Reads	2:1	4	1 bus	2	Inbound bus limited.
Memory Writes	2:1	4	1 bus	2	Outbound bus limited.

7. Signal Description

This chapter describes the external signals of the 970MP microprocessor. It contains a concise description of individual signals, showing behavior when the signal is asserted and negated and when the signal is an input and an output.

Note: A bar over a signal name indicates that the signal is active low. For example, $\overline{\text{CHKSTOP}}$ (checkstop in/out) and $\overline{\text{BYPASS}}$ (PLL bypass). Active-low signals are referred to as asserted (active) when they are low and negated when they are high. Signals that are not active low, such as ADIN[0:43] (address bus signals) are referred to as asserted when they are high and negated when they are low.

The 970MP microprocessor signals are grouped as follows:

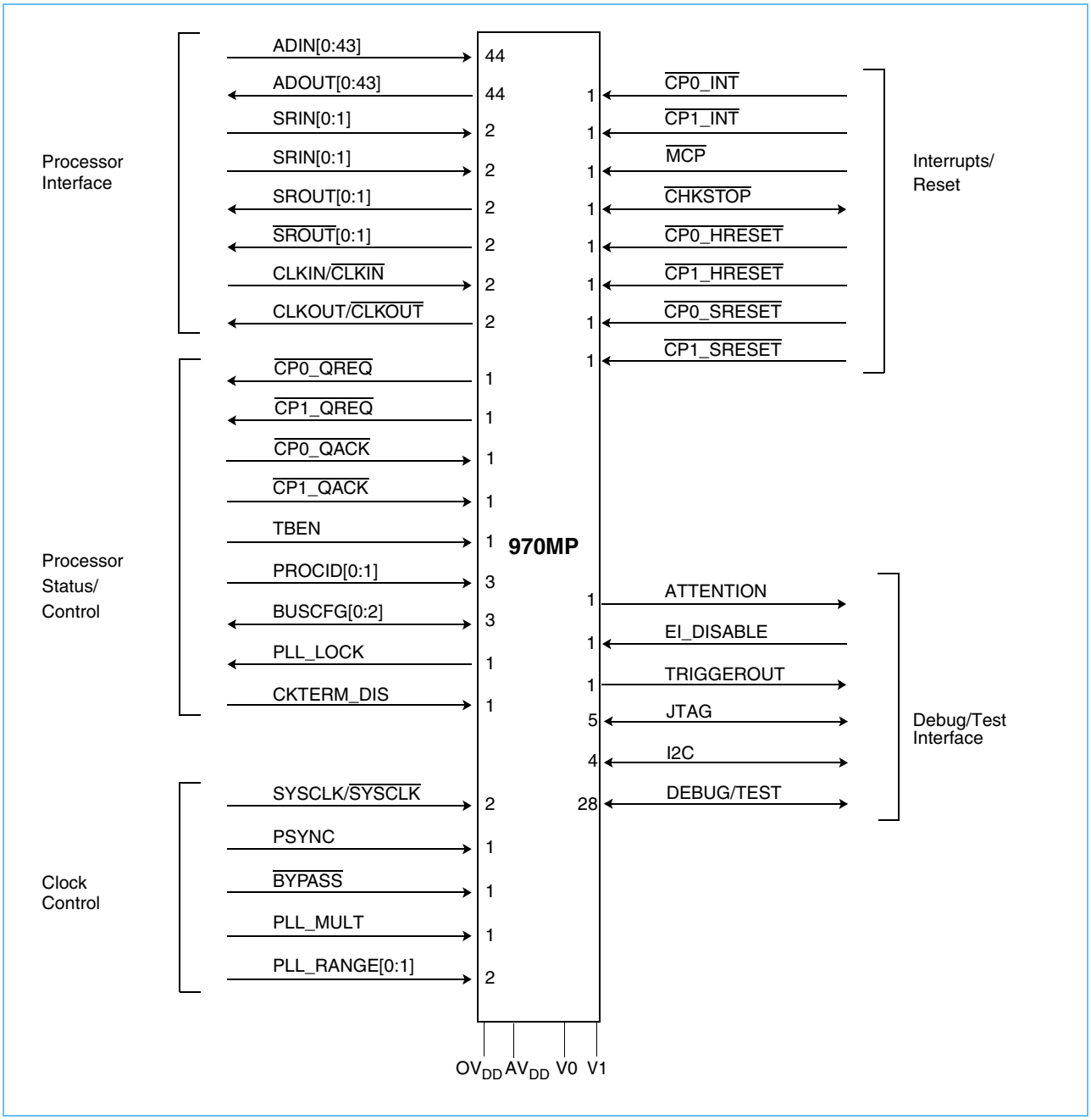
- Processor interface—These signals are used to transfer address, data, and control information between the 970MP microprocessor and a companion chip to provide coherent access to memory and access to memory-mapped I/O.
- Processor status and control—These signals are used to monitor and provide external control of various processor facilities, including the external bus and power management.
- Clock control—These signals determine the system clock frequency. They can also be used to synchronize multiprocessor systems.
- Interrupts/resets—These signals include the external interrupt signal, checkstop[1] signals, and both soft reset and hard reset signals. They are used to interrupt and to reset the processor under various conditions.
- Debug/test interface—The debug/test interface provides a serial interface to the system for performing debug, bring-up, and manufacturing tests. The JTAG (IEEE 1149.1a-1993) interface and the inter-integrated circuit (I^2C) interface provide a serial interface to the system for performing board-level boundary-scan interconnect tests.

1. Hardware has detected a condition that it cannot resolve and which prevents normal operation. It stops executing instructions, responding to interrupts, and so on.

7.1 Signal Configuration

Figure 7-1 illustrates the configuration of the 970MP microprocessor signals, showing how the signals are grouped. A pinout showing pin numbers is included in the *IBM PowerPC 970MP RISC Microprocessor Datasheet*.

Figure 7-1. 970MP Microprocessor Signal Groups



7.2 Signal Descriptions

This section describes individual signals on the 970MP microprocessor, which are grouped as shown in *Figure 7-1 970MP Microprocessor Signal Groups* on page 220. In the following section, "cycle" or "clock" refers to a single bus clock period, which may correspond to one or more internal processor clocks depending on the clock mode programmed for the 970MP microprocessor.

Note: In PLL-bypass mode, the SYSCLK input signal clocks the internal processor directly, the PLL is disabled, and the bus mode is set to whatever bus mode is selected. This mode is intended for factory use only.

7.2.1 Processor Interface

The processor interface provides a high-speed, source-synchronous, point-to-point connection between the 970MP microprocessor and a companion chip. It consists of two unidirectional sets of signals, one to carry outgoing information from the 970MP microprocessor, the other to carry incoming information to the 970MP microprocessor. Each of these two sets of signals consists of a 44-bit bus to transfer logical data with redundancy, a differential clock (two signals), and a 2-bit differential snoop response (four signals).

Chapter 8 provides detailed information about the format and timing of these signals as they are used in the processor interconnect protocol implemented in the 970MP microprocessor.

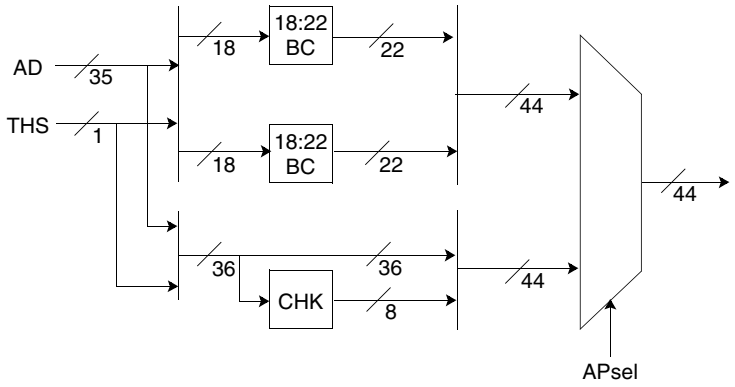
7.2.1.1 Address/Data In (ADIN[0:43])–Input

The address/data input signals carry address, data, and control information from the companion chip to the 970MP microprocessor. The 44 bits of ADIN carry 36 bits of address/data (AD) and transfer-handshake (TH) information plus 8 bits of redundancy.

There are two defined formats for encoding the 36 AD and TH signal lines onto the 44 source-synchronous bus (SSB) signal lines (see *Figure 7-2 Encoding and Selection Logic for the Drive Side of a 970MP Interconnect SSB* on page 222). The first format exploits a balanced coding method (BCM) to maintain an equal number of zeros and ones on the signal lines. During any valid state of the bus, exactly 22 of the signals lines are high and 22 are low. The BCM advantage is that it dramatically improves the signal-to-noise robustness of the bus for high-speed operation at the cost of a few extra signal lines. The BCM can inherently detect a single bit error from any of the 44 signal lines.

The second mode uses 36 of the 44 SSB signal lines for the data transfer. The remaining eight SSB signal lines are used to encode an 8-bit parity value that has sufficient redundancy to detect up to two bit errors across any of the 44 SSB signal lines and correctly identify the bit position of any single bit error.

Figure 7-2. Encoding and Selection Logic for the Drive Side of a 970MP Interconnect SSB



In BCM mode, the 36 inputs are partitioned between two 18-bit balanced code (BC) encoders. In the alternate mode, the 36 bits pass straight through and 8 parity bits are added to the output. A select signal line, called APsel, is programmed through the I^2C interface to select which mode is used.

Timing: The processor interface is source synchronous, meaning that the same clock that launches data on the sending end is transferred with the data and used at the receiving end to capture the data. The interface is run in double data rate (DDR) fashion, with a data transfer on every rising and falling edge of the clock. Because there is no arbitration on this interface, valid data can be transferred on any clock edge. ADIN uses CLKIN as its reference.

7.2.1.2 Snoop Response In (SRIN[0:1], $\overline{SRIN}$[0:1])–Input

The snoop-response input signals carry a 2-bit code from the companion chip to the 970MP microprocessor, indicating the coherency response of the system to an earlier command sent on the ADOUT bus. SRIN and $\overline{SRIN}$ represent a differential pair, such that SRIN carries the snoop response in an asserted high signal level at the same time that $\overline{SRIN}$ carries the same snoop response in an asserted low signal level.

Timing: Same as ADIN.

7.2.1.3 Clock In (CLKIN/$\overline{CLKIN}$)–Input

The CLKIN signal originates in the companion chip and is sent synchronously with the data (ADIN and SRIN) for use in data capture at the receivers in the 970MP microprocessor. This clock is transmitted as a differential pair.

Timing: The clock in signal is derived from the on-chip PLL on the companion chip and synchronized to the *psync* signal, which provides a periodic global reference event. During the initial alignment procedure (IAP) for the processor interface, a rising edge of the clock in signal is identified as corresponding to time zero. Every other rising edge thereafter is a time zero, delimiting the basic unit of time on the bus, in which four beats of data can be transferred.

7.2.1.4 Address Data Out (ADOUT[0:43])–Output

The address/data output signals carry address, data, and control information from the 970MP microprocessor to the companion chip. The 44 bits of ADOUT carry 36 bits of address/data (AD) and transfer-handshake (TH) information plus 8 bits of redundancy, similarly to ADIN.

Timing: Same as ADIN, except that ADOUT uses CLKOUT as its reference.

7.2.1.5 Snoop Response Out (SROUT[0:1], $\overline{\text{SROUT}}$[0:1])–Output

The snoop-response output signals carry a 2-bit code from the 970MP microprocessor to the companion chip, indicating the coherency response of the processor to an earlier reflected command sent on the ADIN bus. SROUT and $\overline{\text{SROUT}}$ represent a differential pair, such that SROUT carries the snoop response in an asserted high signal level at the same time that $\overline{\text{SROUT}}$ carries the same snoop response in an asserted low signal level.

Timing: Same as ADOUT.

7.2.1.6 Clock Out (CLKOUT/$\overline{\text{CLKOUT}}$)–Output

The clock out signal originates in the 970MP microprocessor and is sent synchronously with the data (ADOUT and SROUT) for use in data capture at the receivers in the companion chip. This clock is transmitted as a differential pair.

Timing: The clock out signal is derived from the on-chip PLL on the 970MP microprocessor and synchronized to the *psync* signal, which provides a periodic global reference event. During the IAP for the processor interface, a rising edge of the clock out signal is identified as corresponding to time zero. Every other rising edge thereafter is a time zero, delimiting the basic unit of time on the bus, in which four beats of data can be transferred.

7.2.2 Processor Status and Control

7.2.2.1 Quiescent Request ($\overline{\text{CP0\_QREQ}}$ and $\overline{\text{CP1\_QREQ}}$)–Output

The $\overline{\text{CP0\_QREQ}}$ and $\overline{\text{CP1\_QREQ}}$ signals, along with $\overline{\text{CP0\_QACK}}$ and $\overline{\text{CP1\_QACK}}$, are used for power management on the 970MP microprocessor. The $\overline{\text{QREQ}}$ signals have two distinct uses. When a frequency shift procedure in the power tuning facility is not in progress, assertion of $\overline{\text{CP0\_QREQ}}$ for PU0 ($\overline{\text{CP1\_QREQ}}$ for PU1) indicates that the 970MP processing unit has entered Doze mode, and is prepared to go into Nap (or Deep Nap) mode. This signal remains asserted until the 970MP processing unit returns to Run mode.

When a frequency shift procedure in the power tuning facility is in progress, assertion of $\overline{\text{CP0\_QREQ}}$ for PU0 ($\overline{\text{CP1\_QREQ}}$ for PU1) indicates that the 970MP processing unit is prepared to perform the frequency shift itself. This signal remains asserted until the 970MP processing unit has completed the frequency shift procedure. See *Chapter 9* for more information about frequency shifting in the power tuning facility.

Timing: The $\overline{\text{CP0\_QREQ}}$ and $\overline{\text{CP1\_QREQ}}$ signals can be asserted or negated by the processor at any time.

7.2.2.2 Quiescent Acknowledgment ($\overline{\text{CP0\_QACK}}$ and $\overline{\text{CP1\_QACK}}$)–Input

The $\overline{\text{CP0\_QACK}}$ and $\overline{\text{CP1\_QACK}}$ signals, along with $\overline{\text{CP0\_QREQ}}$ and $\overline{\text{CP1\_QREQ}}$, are used for power management on the 970MP microprocessor. The $\overline{\text{QACK}}$ signals have two distinct uses. When a frequency shift procedure in the power tuning facility is not in progress, assertion of $\overline{\text{CP0\_QACK}}$ for PU0 ($\overline{\text{CP1\_QACK}}$ for PU1) indicates that all bus activity that requires snooping has stopped, and that the 970MP processing unit can enter Nap (or Deep Nap) mode. This signal must be negated whenever bus activity requiring snooping is resumed, or the 970MP processing unit negates $\overline{\text{QREQ}}$.

When a frequency shift procedure in the power tuning facility is in progress, assertion of $\overline{\text{CP0\_QACK}}$ and $\overline{\text{CP1\_QACK}}$ indicates that the rest of the system is prepared to perform the frequency shift itself. This signal remains asserted until the companion chip has completed the frequency shift procedure. See *Chapter 9* for more information about frequency shifting in the power tuning facility.

Timing: The $\overline{\text{CP0\_QACK}}$ signal for PU0 ($\overline{\text{CP1\_QACK}}$ signal for PU1) is asserted in response to assertion of the $\overline{\text{CP0\_QREQ}}$ signal by PU0 ($\overline{\text{CP1\_QREQ}}$ signal by PU1). It can be asserted any time $\overline{\text{QREQ}}$ is asserted, and can be negated at any time.

7.2.2.3 Time-Base Enable (TBEN)–Input

The TBEN input signal can be used in one of two ways, as determined by the value of HID0[19]. When HID0[19] equals '0', the Time-Base Register is incremented and the Decrementer Register is decremented at 1/16th of the full processor frequency whenever TBEN is asserted. These two timer registers maintain their value when TBEN is negated in this mode.

When HID0[19] equals '1', the Time-Base Register is incremented and the Decrementer Register decremented on every rising edge of the TBEN input signal. In this externally clocked mode, the TBEN frequency must not exceed 1/16th the full processor frequency in order to guarantee sufficient sampling of this external signal.

Timing: The TBEN input is asynchronous to the SYSCLK and processor clocks, and can change at any time, subject to the previously stated frequency restriction.

7.2.2.4 Processor ID (PROCID[0:1])–Input

The 2-bit processor ID is used to assign unique IDs to the two 970MP processing units in a system that can have up to eight processors. The PROCID signals are sampled during power-on reset, and the 2-bit value is placed in the second and third lowest-order bits of the Processor ID Register (PIR) of each processing unit. The lowest-order PIR bit is hardwired to a '0' for PU0 and to '1' for PU1.

Timing: These signals should be permanently tied to V_{DD} or GND, as appropriate for the required ID value.

7.2.2.5 Bus Configuration Select (BUSCFG[0:2])–Input

The 3-bit BUSCFG input encodes the processor clock to bus clock ratio. It is used to select the appropriate clock dividers in the 970MP microprocessor in order to generate the required bus clock frequency. Note that not all encodes work with the power tuning facility (see *Chapter 9* for more information). The interpretation of the BUSCFG values can be found in the *IBM PowerPC 970MP RISC Microprocessor Datasheet*.

Timing: These signals should be permanently tied to V_{DD} or GND, as appropriate for the required bus configuration value.

7.2.2.6 PLL Locked (PLL_LOCK)–Output

The PLL_LOCK signal is asserted when the PLL has achieved lock, or when running in bypass mode. The signal is negated otherwise.

Timing: The PLL_LOCK signal can change at any time. The initial maximum latency for the PLL to achieve lock is specified in the *IBM PowerPC 970MP RISC Microprocessor Datasheet*.

7.2.2.7 Clock Receiver Termination (CKTERM_DIS)–Input

The CKTERM_DIS signal allows the internal termination on the SYSCLK and $\overline{\text{SYSCLK}}$ signals to be disabled. When CKTERM_DIS is negated, the clock in signals are terminated. When the CKTERM_DIS signal is asserted, the termination of the clock in signals is removed from the receiver circuit.

Timing: This signal should be permanently tied to V_{DD} or GND, as appropriate for the required clock configuration.

7.2.3 Clock Control

7.2.3.1 System Clock (SYSCLK/$\overline{\text{SYSCLK}}$)–Input

The SYSCLK inputs provide the reference clock from which the on-chip PLL develops the processor mesh clock, as well as the bus clock. The system clock is provided to the processor as a differential pair. The mesh clock frequency is determined by this reference clock and the value of the PLL_MULT input. The bus clock frequency is determined by the mesh clock frequency and the value of the BUSCFG input. See the *IBM PowerPC 970MP RISC Microprocessor Datasheet* for the correspondence between these inputs and the clock frequency ratios.

Timing: See the *IBM PowerPC 970MP RISC Microprocessor Datasheet* for clock specifications.

7.2.3.2 Phase Synchronization (psync)–Input

The *psync* signal provides a synchronization pulse to all processors and companion chips in the system, providing the basis for identifying a periodic time zero event in each chip.

Timing: See the *IBM PowerPC 970MP RISC Microprocessor Datasheet* for clock specifications.

7.2.3.3 PLL Bypass ($\overline{\text{BYPASS}}$)–Input

The $\overline{\text{BYPASS}}$ signal indicates to the processor that the system clock input should be fed directly to the PLL output, bypassing the PLL. This mode of clocking the processor can be used for debugging.

Timing: To bypass during debug, this signal should be tied to GND.

7.2.3.4 PLL Multiplier (PLL_MULT)–Input

The PLL_MULT signal is used to specify the ratio of the full processor mesh frequency to the system clock frequency. See the *IBM PowerPC 970MP RISC Microprocessor Datasheet* for the correspondence between the value of this signal and the clock ratio.

Timing: This signal should be permanently tied to V_{DD} or GND, as appropriate to the required clock configuration.

7.2.3.5 PLL Range Select (PLL_RANGE[0:1])–Input

The PLL_RANGE signal is used to identify the required frequency range of the processor mesh clock. See the *IBM PowerPC 970MP RISC Microprocessor Datasheet* for the correspondence between the value of this signal and the required frequency range.

Timing: This signal should be permanently tied to V_{DD} or GND, as appropriate to the required clock configuration.

7.2.4 Interrupts and Resets

Most system status signals are input signals that indicate when exceptions are received, when checkstop conditions have occurred, and when the 970MP microprocessor must be reset.

7.2.4.1 Interrupt ($\overline{\text{CP0\_INT}}$ and $\overline{\text{CP1\_INT}}$)–Input

The $\overline{\text{CP0\_INT}}$ and $\overline{\text{CP1\_INT}}$ signals provide a means for raising an external interrupt. This exception can be masked by the MSR[EE] bit. When MSR[EE] equals '0', the processing unit will not respond to the assertion of $\overline{\text{INT}}$.

7.2.4.2 Machine Check Interrupt ($\overline{\text{MCP}}$)–Input

The $\overline{\text{MCP}}$ signal provides a means for raising a machine check exception. This exception can be masked by two control bits. If HID0[32] equals '0', the assertion of $\overline{\text{MCP}}$ is ignored. If HID0[32] equals '1', and MSR[ME] equals '1', machine checks are enabled, and the assertion of $\overline{\text{MCP}}$ will result in a machine check exception being taken. If HID0[32] equals '1', and MSR[ME] equals '0', machine checks are disabled, and the assertion of $\overline{\text{MCP}}$ will cause the processor to enter the checkstop state.

Timing: This signal can be asserted at any time, asynchronously to the system clock. Once asserted, the $\overline{\text{MCP}}$ signal must remain asserted for at least two bus clock cycles to ensure that it is recognized.

7.2.4.3 Checkstop ($\overline{\text{CHKSTOP}}$) –Bidirectional

The checkstop signal is both an input and an output signal on the 970MP microprocessor.

Checkstop ($\overline{\text{CHKSTOP}}$) –Input

The checkstop input signal provides a means for external initiation of a checkstop.

Timing: This signal can be asserted at any time, asynchronously to the system clock.

Checkstop (CHKSTOP) –Output

The checkstop output signal indicates that the processor has entered the checkstop state.

Timing: This signal can be asserted at any time.

7.2.4.4 Hard Reset (CP0_HRESET and CP1_HRESET)–Input

The CP0_HRESET signal provides a means for resetting PU0 and initiating the power-on-reset sequence for PU0. The CP1_HRESET signal provides a means for resetting PU1 and initiating the power-on-reset sequence for PU1.

Timing: This signal can be asserted at any time, asynchronously to the system clock.

7.2.4.5 Soft Reset (CP0_SRESET and CP1_SRESET)–Input

The CP0_SRESET and CP1_SRESET signals provide a means for external initiation of the soft (or warm) reset. When CP0_SRESET is asserted, the PU0 responds by taking a system reset exception. When CP1_SRESET is asserted, the PU1 responds by taking a system reset exception.

Timing: This signal can be asserted at any time, asynchronously to the system clock.

7.2.5 Debug/Test Interface

7.2.5.1 Attention (ATTENTION)–Output

ATTENTION is an output signal from the 970MP microprocessor to the JTAG debugger, used in debug mode. I^2C SCOM commands are sent directly to PSCOM and do not go through the JTAG TAP engine. Therefore, when Attention is active, a SCOM read/write command will not be acknowledged with the standard I^2C acknowledgment (ACK) pulse because it is not a primitive test access port (TAP) command.[1]

7.2.5.2 Processor Interface Disable (EI_DISABLE)–Input

Turns off elasticity in the processor interface bus.

7.2.5.3 Trigger Out (TRIGGEROUT)–Output

TRIGGEROUT is an output signal used to indicate that internal trace collection has begun.

7.2.5.4 JTAG Signals

The IEEE 1149.1 defines a five-wire interface called a test access port (TAP) for communicating with the boundary scan architecture. The five JTAG signals are: TDI, TDO, TMS, TCK, and TRST.

1. Primitive TAP commands are those that scan the IR or DR in the JTAG engine.

Test Clock (TCK)–Input

TCK is a JTAG test clock, which is separate from the system mesh clock. The TCK only controls the test access port functions (20 or 30 latches). SYSCLK must always be active to control the interfaces. The rising edge causes TMS and TDI to be sampled by the Access macro.

Test Data In (TDI)–Input

TDI is a JTAG serial input used to feed test data and test access port instructions.

Test Data Out (TDO)–Output

TDO is a JTAG serial output used to extract data from the chip under test control.

Test Mode Select (TMS)–Input

TMS is a JTAG select signal used to control the operation of the JTAG state machine. The value of TMS during a rising edge of TCK causes a state transition in the TAP controller.

Test Logic Reset ($\overline{\text{TRST}}$)–Input

$\overline{\text{TRST}}$ is an asynchronous JTAG signal used to reset the JTAG state machine. The $\overline{\text{TRST}}$ signal ensures that the JTAG logic does not interfere with the normal operation of the chip. The $\overline{\text{HRESET}}$ signal performs the function of $\overline{\text{TRST}}$ internally.

7.2.5.5 I^2C Signals

The 970MP I^2C bus conforms to the standard-mode timing specification and does not support high-speed or fast-mode timing. The 970MP microprocessor has the following I^2C signals:

- I^2C Signal Clock ($\overline{\text{I2CCK}}$)–I^2C signal clock is both an input and output signal pin.
- I^2C Interface Data ($\overline{\text{I2CDT}}$)–I^2C interface data is both an input and output signal pin.
- I^2C Interface Go (I2CGO)–I2CGO is an asynchronous, open-drain output signal used to prevent access collisions between JTAG and I^2C. If the level of the interface is low, only JTAG should access the 970MP. I^2C can use the interface if the level is high.
- I^2C Select (I2CSEL)–I2CSEL controls the use of the mutually exclusive I^2C or JTAG bus. When asserted, the I^2C bus can be used. Otherwise, the JTAG bus can be used.

7.2.6 Voltage and Ground

The 970MP microprocessor provides the following connections for power and ground:

- OV_{DD}—The OV_{DD} signal provides the supply voltage connection for the drivers and receivers.
- AV_{DD}—AV_{DD} is a power signal that drives the analog sections of the PLL. See the *IBM PowerPC 970MP RISC Microprocessor Datasheet* for information about how to use this signal.
- V0—The V0 (V_{DD}) signal provides the supply voltage connection for processor core 0 and the common logic.
- V1—The V1 (V_{DD}) signal provides the supply voltage connection for processor core 1.

8. Processor Interconnect Bus

The IBM PowerPC 970MP RISC Microprocessor Processor Interconnect is a bus architecture providing high-speed, high-performance interconnections for processors, I/O devices, memory subsystems, and bridge chips. This bus architecture provides a forward-looking, general use, yet cost-effective solution for designing high-performance IBM PowerPC systems.

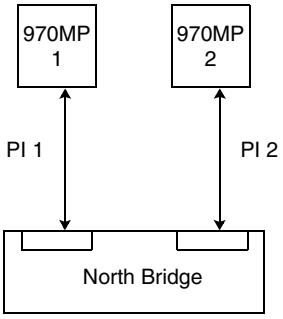
At the heart of the processor interconnect bus is a set of unidirectional, point-to-point bus segments, a new design selected to achieve maximum data transfer rates. The bus segments include two 35-bit address/data segments (one in each direction), two 1-bit transfer-handshake segments, and two 2-bit snoop-response segments. New features include:

- Pipelined transactions for reading and writing data and maintaining cache coherency
- Packet protocols for data sharing, data synchronization, and cache snooping
- True split transactions, enabling the master and slave to simultaneously conduct different transactions with each other
- Wave pipelining to exploit maximum data bandwidth at the electrical interface

The unidirectional segments are the basis for supporting the features above. These buses are point-to-point connections, carry their own local clock signal (source synchronous), and require no arbitration. Error detection mechanisms exist for all bus segments.

There are many possible configurations that incorporate different numbers of processors, I/O interfaces, and memory bandwidth, and meet different speed, cost, and power requirements. *Figure 8-1* shows an example of a configuration with two 970MP microprocessors.

Figure 8-1. Processor Interconnect Bus Configuration with Two 970MP Microprocessors



The remainder of this section specifies the processor interconnect architecture targeting a dual processor, dual-ported North Bridge configuration, as shown in *Figure 8-1*. Using two processor interconnect ports on the North Bridge enables direct connection of two 970MP microprocessors.

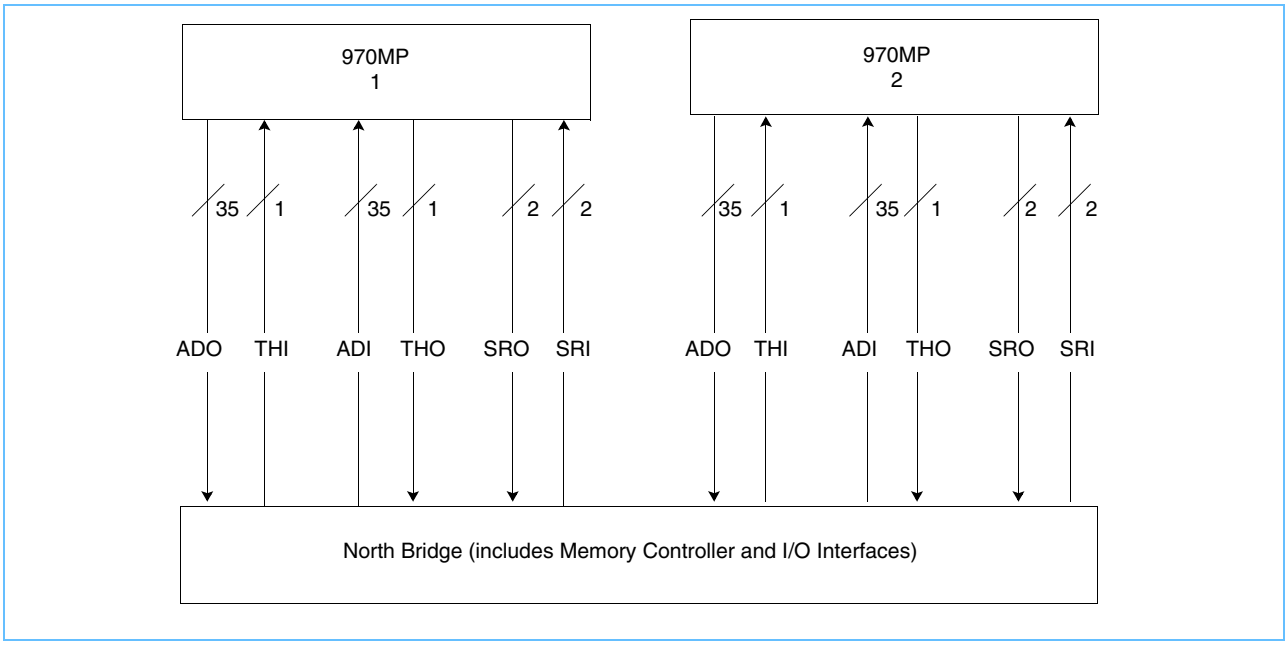
8.1 Overview

The processor interconnect bus consists of a set of unidirectional, point-to-point bus segments for maximum data transfer rates. No bus-level arbitration is required. An address/data (AD) bus segment, a transfer-handshake (TH) bus segment, and a snoop-response (SR) bus segment exist in each direction, outbound and inbound. *Figure 8-2* shows two 970MP microprocessors connected to a North Bridge using two processor interconnect buses.

This section frequently uses the terms "packet," "beat," "master," and "slave." The usage conventions of these terms are as follows:

- Data is transferred across a bus in beats from master to slave. A beat is a timing event relative to the rising or falling edge of the clock signal. Nominally there are two beats per clock cycle (one for the rising edge and one for the falling edge).
- A packet is the fundamental protocol data unit for the processor interconnect bus. A non-null packet consists of an even number of data elements that are sequentially transferred across a source-synchronous bus at the rate of one element per bus beat. The number of bits in each data element equals the width of the bus. Packets are used for sending commands, reading and writing data, maintaining distributed cache coherency, and transfer-protocol handshaking.
- A sender or source of packets for a bus segment is called a master and a receiver or recipient is called a slave. For example, on an outbound processor bus segment, the North Bridge is the slave and the processor is the master. On an inbound processor bus segment, the North Bridge is the master and the processor is the slave.

Figure 8-2. Two Microprocessors Connected to a North Bridge



8.1.1 Packets

Four basic packet types are defined: null packets, command packets, data packets, and transfer-handshake packets. Non-null packet lengths are always an even number of beats.

Null packets are sent across the address/data bus. For the null packet, all bits are zero. Null packets are ignored by slave devices.

Command packets are sent across the address/data bus. There are three types of command packets: read-command packets, write-command packets, and coherency-control packets.

Data packets are also sent across the address/data bus. There are two types of data packets: read-data packets and write-data packets. A write-data packet immediately follows a write-command packet. A read-data packet is sent in response to a read-command packet or a cache-coherency snoop operation. A data read header contains the address of the command, the command type, and transfer details.

Transfer-handshake packets are sent across the transfer-handshake bus. This packet is issued to confirm receipt and indicate the condition of the received command packet or data packet. Condition encoding includes Acknowledgment, Retry, Parity Error, or Null/Idle. A transfer-handshake packet is two beats in length.

See *Section 8.2 Packet Transfer Protocol* on page 237 for a detailed description of these four packet types.

8.1.2 Bus Segments

An AD bus segment, a TH bus segment, and an SR bus segment exist in each direction, outbound and inbound. *Table 8-1* and the following subsections further describe these signals.

Table 8-1. Processor Interconnect Signal Description

Signal Names	Signal Lines	Mnemonic	Description
Address/Data Out	35	ADO	Address or data and control information
Transfer Handshake Out	1	THO	Acknowledgment packet for command and data packets received on the address/data in bus
Snoop Response Out	2	SRO	Snoop coherency response from the processor
Address/Data In	35	ADI	Address or data, and control information
Transfer Handshake In	1	THI	Acknowledgment packet for command and data packets received on the address/data out bus
Snoop Response In	2	SRI	Accumulated snoop coherency response from the North Bridge

8.1.2.1 Address/Data Bus Segment

The address/data bus is used to transfer both command packets (containing control information) and data packets (containing the data to be transferred). The address/data bus consists of one 35-bit outbound address/data (ADO) bus segment and one 35-bit inbound address/data (ADI) bus segment.

Commands are issued to the bus as 2-beat packets. A read-data packet consists of a 2-beat header followed by the data payload. The number of beats issued with a data transfer depends on the size of the total transfer. Data payload is issued to the bus in even multiples of 4-byte wide data beats. Included in the packet is a bit for special system support and a data error bit.

8.1.2.2 Transfer-Handshake Bus Segment

The transfer-handshake bus sends transfer-handshake packets, which confirm that command or data packets were received on the address/data bus. The transfer-handshake bus consists of one 1-bit outbound transfer-handshake (THO) bus segment and one 1-bit inbound transfer-handshake (THI) bus segment. Every device issuing a command packet, data packet, or reflected command packet to the address/data bus receives a transfer-handshake packet through the transfer-handshake bus some fixed number of beats after issuing the command or data packet.

Each transfer-handshake bus segment sends transfer packets for command and data packets transferred in the opposite direction. That is, the outbound transfer-handshake bus sends acknowledgment packets for the command and data packets received on the inbound AD bus. There is no dependency or relationship between packets on the outbound address/data bus and the outbound transfer-handshake bus.

A transfer-handshake packet might result in a command packet being reissued to the bus because a data buffer in the command queue is full. IBM suggests that the North Bridge implement queues that are deep enough to minimize the impact of command packet retries on system performance.

A transaction remains active until it has passed all response windows. For write transactions, this includes the last beat of the data payload. Since commands might be retried for queue or buffer full conditions, transactions that must be ordered cannot be simultaneously in the active state.

A write transaction issued by the processor can be retried. The slave issues two transfer-handshake packets for a write transaction. The first packet is for the write-command packet and the second for the write-data packet.

For read transactions, the processor will not retry inbound (memory to processor) transfers. Reflected commands (that is, snoop requests inbound from the North Bridge to the processor) cannot be retried. This is necessary to ensure a fixed snoop window is maintained.

8.1.2.3 Snoop-Response Bus Segment

The snoop-response bus supports global snooping activities to maintain cache coherency. A processor uses this bus to respond to a reflected command packet received on the ADI bus. The snoop-response bus consists of one 2-bit, outbound snoop-response (SRO) bus segment and one 2-bit, inbound snoop-response (SRI) bus segment. The bus segments can detect single bit errors.

A snoop response begins when a processor receives a reflected command packet on the ADI bus. The processor provides a snoop response reporting the coherency status of the request received on the ADI bus segment. The North Bridge gathers snoop responses from all processors and sends the accumulated snoop response on the SRI bus segments concurrently to all processors.

8.1.3 Transactions

Three transaction types are defined: read, write, and command-only. *Section 8.4 Bus Transactions* on page 253 describes the transactions in detail. The following subsections show the sequence of operations for these transaction types.

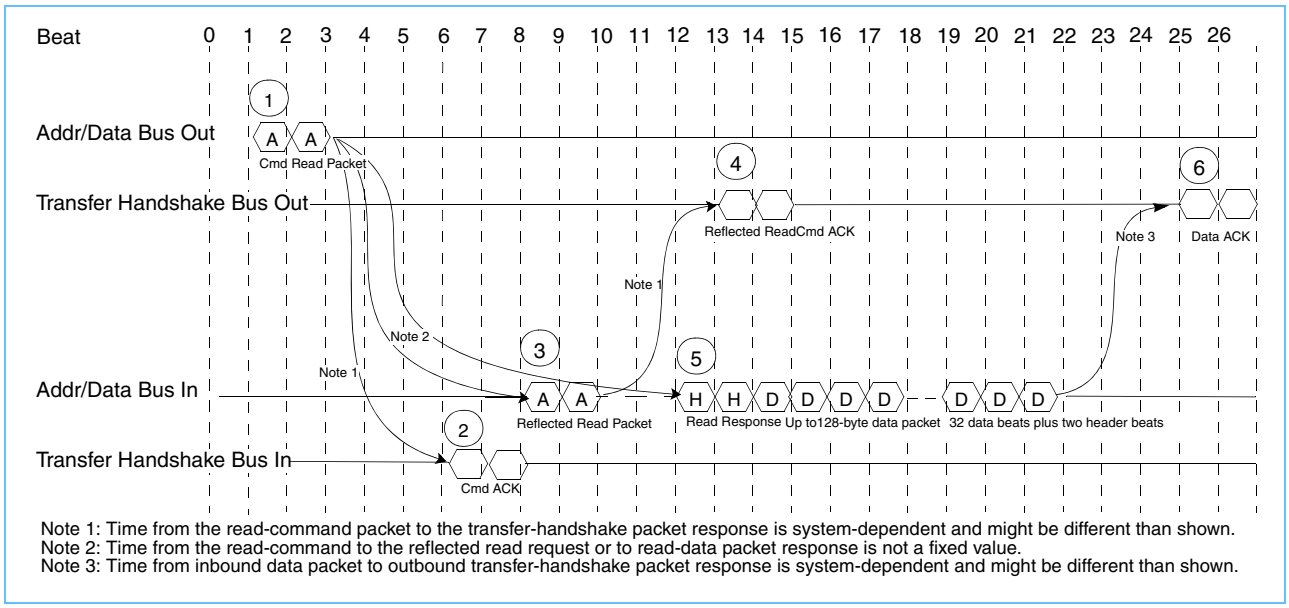
8.1.3.1 Read Transaction

Figure 8-3 shows the sequence of operations for a read transaction.

1. The master (requesting processor) issues a read-command packet on the ADO bus segment to request a full or partial cache line of data from the slave (North Bridge).
2. The slave sends a transfer-handshake packet to the master on the THI bus segment.
3. For cache-coherency purposes, the slave reflects the read-command packet on the ADI bus segment to all processors.
4. Each processor sends a transfer-handshake packet on the THO bus segment to the slave in response to the reflected read-command packet.
5. The slave sends the read-data packet on the ADI bus segment to the master.
6. The master sends a transfer-handshake packet on the THO bus segment to the North Bridge in response to the read-data packet.

The read-data packet transfer ranges from 4 to 34 beats. The first two beats transferred are a header containing the master's tag and data packet size. The data payload portion must be transmitted in sequence with the critical word first. A command packet might then be interjected into the data payload portion on an even-beat boundary.

Figure 8-3. Read Transaction Timing Diagram



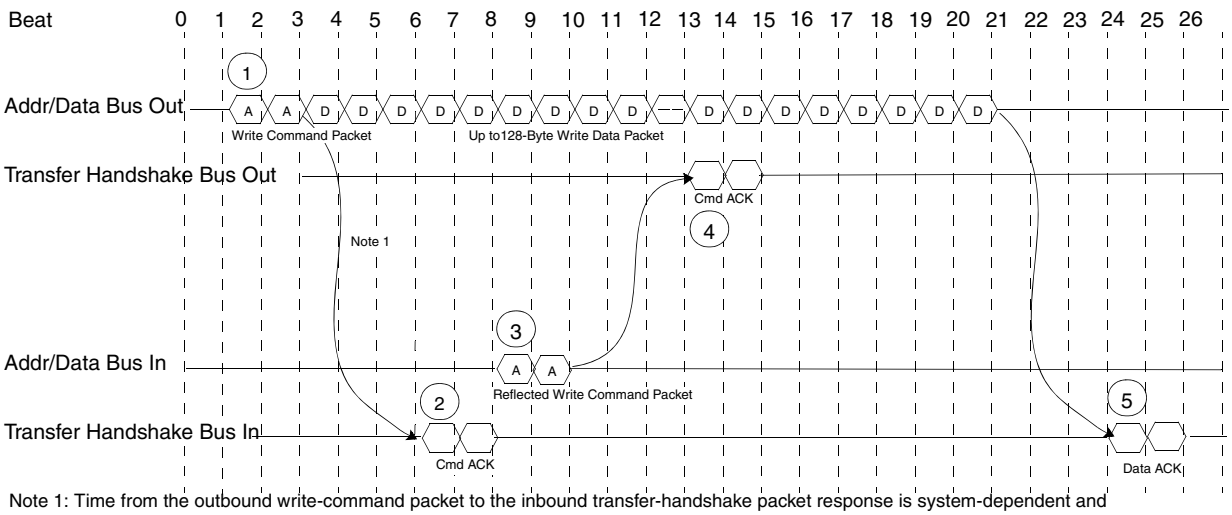
Note 1: Time from the read-command packet to the transfer-handshake packet response is system-dependent and might be different than shown.
Note 2: Time from the read-command to the reflected read request or to read-data packet response is not a fixed value.
Note 3: Time from inbound data packet to outbound transfer-handshake packet response is system-dependent and might be different than shown.

8.1.3.2 Write Transaction

A processor initiates a write transaction to store either a full or partial cache line of data to memory or to an I/O device. A write transaction consists of a command packet immediately followed by a data packet on the master's ADO bus segment. The data must be issued to the address/data bus segment in consecutive beats, but can be paused on an even beat to issue a command packet for a read operation. A write-command packet cannot be interjected into a write-data packet transfer. *Figure 8-4* shows the sequence of operations for a write transaction.

1. The master (requesting processor) issues a write-command packet on the ADO bus segment to write a full or partial cache line of data. The write-command packet is immediately followed by a write-data packet.
2. The slave (North Bridge) sends a transfer-handshake packet on the THI bus segment in response to the write-command packet.
3. For cache-coherency purposes, the slave reflects the write-command packet on the ADI bus segment to all processors.
4. Each processor sends a transfer-handshake packet on the THO bus segment to the slave in response to the reflected write-command packet.
5. The slave sends an acknowledgment packet on the THI bus segment to the master in response to the write-data packet.

Figure 8-4. Write Transaction Timing Diagram



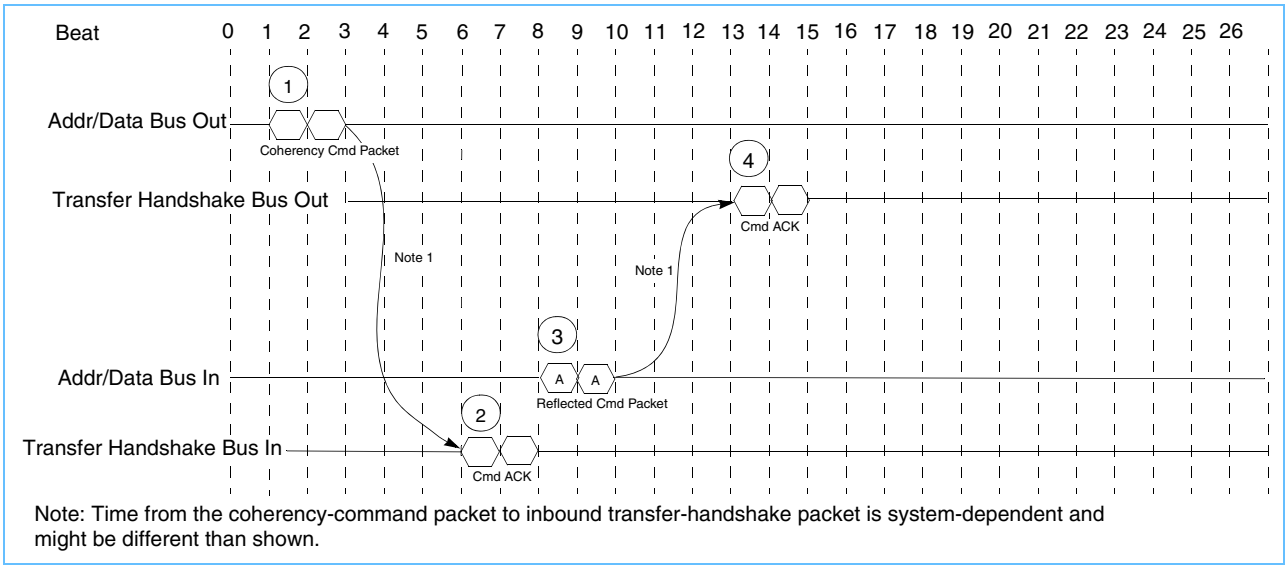
Note 1: Time from the outbound write-command packet to the inbound transfer-handshake packet response is system-dependent and might be different than shown.

8.1.3.3 Command-Only Transaction

Figure 8-5 shows the sequence of operations for a command-only transaction.

1. The master (requesting processor) issues a command packet to the slave (North Bridge) on the ADO bus segment.
2. The slave sends a transfer-handshake packet to the master on the THI bus segment in response to the command packet.
3. For cache-coherency purposes, the slave reflects the command packet on the THI bus segment to all processors.
4. Each processor sends a transfer-handshake packet on the THO bus segment in response to the reflected command packet.

Figure 8-5. Command-Only Transaction Timing Diagram



Note: Time from the coherency-command packet to inbound transfer-handshake packet is system-dependent and might be different than shown.

8.1.4 Memory and Cache Coherency

8.1.4.1 Physical Memory Size

The PowerPC Architecture supports a maximum physical address bus of 64 bits. The processor interconnect specification limits the memory addressing to 42 bits. This allows for a maximum address space of 4 terabytes (TBytes).

8.1.4.2 Coherency Protocol

Coherency is maintained using global snoops of all command packets by reflecting command packets from the North Bridge to the processor. The snoop-response bus is used exclusively for this purpose. This bus consists of two unidirectional 2-bit bus segments per processor port, and is used to source response out and receive response in. Responses are sourced at a configurable time after the global snoop. The response in is sampled at a later time, also configurable. The snooping protocol is detailed in *Section 8.3 Snoop Responses* on page 248.

8.1.4.3 Coherency Block Size

The cache line is the smallest increment of memory over which coherency information is maintained. This bus can support 32-byte, 64-byte, and 128-byte coherency block sizes. The coherency block size is determined by the target processor. All bus attachments must support this coherency block size for uniform operation. The I/O must be capable of transferring less than or equal to, but not greater than, the coherency block size during direct memory access (DMA) transfers to and from coherent memory.

8.2 Packet Transfer Protocol

This section defines packet protocols for data sharing, data synchronization, and cache snooping. The processor interconnect defines four basic packet types: null packets, command packets, data packets, and transfer-handshake packets.

8.2.1 Command Packet Definition

The command packet transfer protocol specifies how addresses are passed between bus devices. Due to the narrow width of the bus, this transfer takes two bus beats to complete, thereby allowing one command packet every two beats.

The command packet consists of a memory address, command type, command size, and command tag. The command packet is identified on the address/data bus by the detection of the packet start signal and a packet-type encoding for a command packet. *Table 8-2* shows the bit definitions for the address/data bus during a command-packet transfer.

Table 8-2. Command Packet Description

Beat	Bits	Description
1	0:1	'10' (Address valid decode).
1	2:6	Transfer Type (0:4).
1	7:15	Transfer Tag (0:8).
1	16:17	Address Modifiers I/S, M (1:2).
1	18:34	Address (42:58). 17 bits of the 42-bit address.
2	0:1	'10' (Address valid decode).
2	2	Address Modifier W/N (0) .
2	3:6	Transfer Size (0:3).
2	7:26	Address (22:41). Most-significant 20 bits of the 42 address bits.
2	27:29	Address Modifiers G, R, P (3:5).
2	30:34	Address (59:63). Least-significant 5 bits of the 42 address bits.

Note:
W: write through, M: memory coherent, N: intervention, A: atomic, R: rerunning, I: cache inhibited, S: noncacheing coherent read, P: pipelined snoops, G: guarded read.

8.2.1.1 Address Modifiers

Bits 16:17 of beat one, and bit 2 and bits 27:29 of beat two of a command packet contain the address modifier bits. These bits further describe the type of command packet. In some cases, they must be decoded along with the Transfer Type bits to determine the operation.

Table 8-3 Transfer Type Encoding on page 239 shows when these bits are used to modify transactions, what the modification is, and what the values are when they are hard coded. Under certain conditions, some bits might be sourced from the page table WIM bits ("W" stands for write through, "I" for cache inhibit, "M" for memory coherence).

Address Modifier[0]

The AM[0] bit, when indicated as a W, means that write through is wanted. When the bit is '1', it means that the data for a write transaction is to be forwarded all the way to system memory or a memory-mapped device. When the bit is '0', the data must be forwarded at least one cache level toward memory. This bit is normally, but not always, sourced from the page table.

On a read operation, when indicated as an N, this bit defines whether the master can support intervention on this request. If intervention is enabled (N equals '1'), then the transfer size must be the coherency block size. (A snooper might not intervene if this bit is reset, and might intervene if it is asserted.)

Address Modifier[1]

The AM[1] bit, when indicated as an I, indicates Cache-Inhibit status. If the bit is '1' in a write-command packet, it means that the data should not be cached downstream from this processor. When indicated as an S and the bit is a '1' in a read-command packet, it means that the requesting processor will not cache the data when received, and memory (or an intervening cache) might still retain the current coherency status.

Address Modifier[2]

The AM[2] bit, when indicated as an M, is always used as the memory coherent indicator or snoop request signal. If this bit is '0', the horizontal coherency snoopers ignore this transaction, meaning memory is not coherent or this is a transaction that snoopers do not need to look at (vertical caches need to snoop all snoop-response [SResp] enabled transactions regardless of the M bit).

Note: This bit should be defined consistently for future transactions that might be architected, as snoopers will not see any transaction where M equals '0'. This bit is frequently sourced from the page table WIM bits when indicated as an M, but at other times it is hard coded so snoopers see the transaction. For example, it might be set by an I/O adapter for coherent I/O.

Address Modifier[3]

The AM[3] bit is used to further define operations. For example, it is used to indicate a write-with-kill versus a write-with-clean. When indicated as a G, it is used to indicate a guarded read.

Address Modifier[4]

The AM[4] bit, when indicated as an R, means that this transaction has already been issued to the bus once, and is now being reissued.

Implementation Note: The bit should be set to zero in current implementations of the architecture, to remain compatible with potential architecture extensions.

Address Modifier[5]

The AM[5] bit, when indicated as a P, means that this transaction can be pipelined for snoop requests and responses. If P is '0', then command packets are reflected one at a time after the snoop response for previous command packets are seen by all processors.

8.2.1.2 Transfer Type Field

The transfer type (TType) field indicates the type of command packet that was issued to the bus. The valid transfer types defined by the processor interconnect bus are shown in *Table 8-3*. Both the processor and the North Bridge must support all commands listed in *Table 8-3*. I/O devices support only a limited subset of the commands.

Table 8-3. Transfer Type Encoding

Address Modifiers (WIMGRP)	TType Binary	TType Hex	Bus Operation	Code	Address Format	Data Payload	Comments
XXMXRP	00000	00	Clean	CL	Mem	N	M = '1' normally
WIMXRP	00010	02	Write with Flush	WNB	Mem	Y	M = '1' normally
XXMXRP	00100	04	Flush	FL	Mem	N	M = '1' normally
WXM0RP	00110	06	Write with Kill	WBK	Mem	Y	W = 'X' if from a I/O bridge
WXM1RP	00110	06	Write with Clean	WBC	Mem	Y	W = '1', I = 'X', M = '0'
XXMXRP	01000	08	SYNC	SY	Tag	N	
NSMGRP	A1010	0A,1A	Read	RD	Mem	N	S = '1' means RWNITC
XXMXRP	01100	0C	DKill	DK	Mem	N	M = '1'
NXMXRP	A1110	0E,1E	RWITM	RWITM	Mem	N	I = 'X', normally M = '1'
XXMX0P	10000	10	EIEIO	EI	Tag	N	M = '0'
XXMXXX	10100	14	Reserved				M = '0'
XXMX0P	11000	18	TLBIE	TI	Tag	N	M = '0', P = '0'
XXMXXX	11100	1C	Reserved				M = '0'
XXMXRP	00001	01	LARX-Reserve	LR	Mem	N	M = '0'
XXMXRP	A0011	03,13	DClaim	DC	Mem	N	M = '1'
XXMXXX	001X1	05,07	Reserved				M = '0'
XXMXRP	01001	09	TLBSYNC	TS	Tag	N	M = '0'
XXMXXX	01X11	0B,0F	Reserved				M = '0'
XXMX0P	01101	0D	IKill	IK	Mem	N	M = '1' normally, P = '0'
XXMXXX	10001	11	Reserved			A	M = '0'
XXMXXX	10010	12	Reserved			N	M = '0'
XXMXRP	10101	15	Deallocate Dir Tag	DDT	Mem	A	M = '0'
XXMXXX	1011X	16,17	Reserved for customers				M = '0'
XXMXXX	110X1	19,1B	Reserved				M = '0'
XXMX0P	11111	1F	Null	NUL	None	N	M = '0'

Note:

W: write through, M: memory coherent, N: intervention, A: atomic, R: rerunning, I: cache inhibited, S: noncacheing coherent read, P: pipelined snoops, G: guarded read, X: drive '0' when driving signal and don't care when receiving the signal.

Address Field

The address field contains the address associated with the command packet. This field is defined to be 42 bits wide.

Transfer Size Field

The transfer size field indicates the size of the data packet associated with the command packet. For command packets that do not have a data packet associated with them, this field is undefined. *Table 8-4* defines the encoding for the transfer size field for the commands that require a data packet.

Table 8-4. Transfer Size Encoding

Transfer Size	Description	Number of Data Beats
0000	8 Bytes	2
0001	1 Byte	2
0010	2 Bytes	2
0011	3 Bytes	2
0100	4 Bytes	2
0101	5 Bytes	2
0110	6 Bytes	2
0111	7 Bytes	2
1000	128 Bytes	32
1001	16 Bytes	4
1010	32 Bytes	8
1011	Reserved	
1100	64 Bytes	16
1101	Reserved	
1110	Reserved	
1111	Reserved	

Transfer Tag

Command packets contain a 9-bit transfer tag used to link a command with data. This field is valid for all transactions to the bus and contains a number (generated by the processor) to identify the read-data packet on a read transaction and the write-data packet for a write transaction. Explicit tagging of command and data packets allows a bus device to have multiple concurrent outstanding transactions that require a data packet. This means that read-data packets can appear out-of-order on the bus so that transactions can complete when data is available as opposed to returning all data packets in the order the commands were issued. In addition, the tag can be used to reference the response back to a command in an internal queue of a bus device. There must only be one outstanding transaction referred to by a tag at any time.

Tag Deallocation For Read Operations

Read transactions use the tag field to identify incoming read-data packets that are associated with the transaction. Once a tag is assigned to a read transaction, it cannot be reissued until all the read data has been received.

Tag Deallocation For Store, Castout, and Push Operations

Address/data command tags remain active until a clean global snoop response is received.

8.2.1.3 Tag Definition

Table 8-5 defines the 9-bit tag that is sent with a command or read-data packet.

Table 8-5. Tag Definition

Bits	Field Name	Description
0:3	Master number	Master number (one must be reserved for the North Bridge)
4:8	Master tag	Tag (one of 32) assigned to the master's requesting resource

Interjecting Command Packets

Data transfers on the bus are either write-data packets issued with a write-command packet, or read-data packets. These transfers consist of multiple data beats. When a transfer contains multiple beats of data payload transfer, a command packet might be interjected on an even-beat boundary. This feature allows new transactions to be started without having to wait for a long multi-beat data transfer to complete. This protocol allows read-command packets and coherency-control packets to be interjected. Write, castout, push, partial write operations, or other data packets cannot be interjected into a multiple-beat data transfer.

8.2.1.4 Command Pacing

It is possible for the processor to issue command packets at a rate faster than the slave can accept. The slave must then retry the packets so the commands are not lost. This is undesirable because of the additional bus bandwidth consumed for the retried commands. The North Bridge should implement queues that are sufficiently deep to minimize the impact of command packet retries on system performance. This scenario assumes the slaves can handle consecutive data packets, which requires the data buffering to be run at least at the bus clock speed. To avoid this situation, a command pipeline delay parameter, COMPACE, is defined for the bus.

The command pipeline delay parameter is a 4-bit field that is programmed into each bus master to indicate the number of bus beats of delay that must be placed between each command packet on the bus. The delay is in bus beats (assumed to be even). The allowable range of values for COMPACE and related processor delay parameters can be found in *Table 11-1 Programmable Delay Parameters* on page 371. *Table 11-1* also lists North Bridge delay parameters and typical values that these parameters might take on. See *Section 11.2.2 Configurable Parameters* on page 369 for additional information about these configurable delay parameters.

Note: This does not restrict the use of intervening bus beats for data packets.

8.2.2 Data Packet Definition

The data packet transfer protocol specifies how data is passed between bus devices. A data packet is defined as an even-numbered beat transfer on the address/data bus. A write-data packet immediately follows a write-command packet. It is identified on the bus by the data valid decode. A read-data packet has a 2-beat header that includes the tag and the data size. Typically, read-data packets are sent from the North Bridge to a processor. However, during intervention, a processor can send a read-data packet to the North Bridge.

A data packet of the minimum size consists of 8 bytes of data and the data error signal (DERR) to validate the data. Up to 16 pairs of data beats are used to transfer a cache line. *Table 8-6* shows the bit definitions for the read-data packet header on the address/data bus. *Table 8-7* shows the bit definitions for the address/data bus during a data transfer.

Table 8-6. Read-Data Packet Header Description

Beat	Bits	Description
1	0:1	'11' (Data and Address valid decode).
1	2:6	Reserved.
1	7:15	Transfer Tag (0:8).
1	16:18	Reserved.
1	19:22	Responder or Intervener ID.
1	23:34	Reserved.
2	0:1	'11' (Data and Address valid decode).
2	2	Reserved.
2	3:6	Transfer Size (0:3).
2	7:34	Reserved.

Table 8-7. Data Beat Description

Beat	Bits	Description
1	0:1	'01' (Data valid decode).
1	2:33	Next consecutive four bytes of the data packet.
1	34	Data error signal (DERR) indicates an off-bus data error; full data transfer is invalid.
2	0:1	'01' (Data valid decode).
2	2:33	Next consecutive four bytes of the data packet.
2	34	Reserved.

8.2.2.1 Two-Beat Transfers

The processor interconnect supports data transfers of varying lengths. Since the payload portion of all data packets must be at least two beats, a transfer of less than 8 bytes must be padded with additional data to fill the 8-byte minimum transfer size. The data on the bus must be address aligned so a request must be separated into two requests if an 8-byte address boundary is crossed. A master can transfer from 1 to 8 bytes of data during this operation. Data is returned in the original memory order. *Table 8-8* shows the address restrictions for transfers of 1 to 8 bytes.

Table 8-8. Two-Beat Data Transfers

Starting Address[61:63]	Byte Lanes																Data Size
	00	01	02	03	04	05	06	07									
000 – 111	x	x	x	x	x	x	x	x									1 Byte
000, 010, 100, 110	x		x		x		x										2 Byte
000	x																3 Byte
000, 100	x				x												4 Byte
000	x																8 Byte

Note:

1. 'x' is a valid starting position.
2. The operand may not cross a double word boundary.

8.2.2.2 Multi-Beat Transfers

The processor interconnect supports multiple-beat data transfers that are 16, 32, 64, and 128 bytes in length. All such requests for writes and reads that are less than a full coherency block (128 bytes) must be aligned to an address boundary equal to the size of the transfer. For read-data transfers that are a full coherency block, data is returned with the critical 16 bytes first, followed by the remaining data in an interleaved burst order. The resulting data transfer is a block of data that is aligned to the size of the request.

Data Transfer Format

On read data packet transfers that are a full coherency block, the order of the returned data words depends on the address that was specified inside the command packet. Each block of the read data packet is transferred in a sequence of 32-byte data beats. Data ordering is based on the block size. Within a word, data is always transferred in-order starting with the most-significant byte and ending with the least-significant byte.

Partial write commands with transfer sizes less than 8 bytes cannot cross an 8-byte boundary. All write commands (including write, castout, push, and partial write) with transfer sizes of 8 bytes or more must be aligned on an address boundary equal to the size of the transfer.

Table 8-9. Packet Ordering for 128-Byte Interleaved Packets on 32-Byte Boundaries

Address (57:59) 128-Byte	Packet Order Viewed at 16-byte Read Data Transfer	Packet Order Viewed at 32-byte Read Data Transfer
000	0 1 2 3 4 5 6 7	0 1 2 3
001 (Not Valid)	~~1 0 3 2 5 4 7 6~~	
010	2 3 0 1 6 7 4 5	1 0 32
011 (Not Valid)	~~3 2 1 0 7 6 5 4~~	
100	4 5 6 7 0 1 2 3	2 3 0 1
101 (Not Valid)	~~5 4 7 6 1 0 3 2~~	
110	6 7 4 5 2 3 0 1	3 2 1 0
111 (Not Valid)	~~7 6 5 4 3 2 1 0~~	

Table 8-10. Packet Ordering for 32-Byte Interleaved Packets

Address (59:60)	Packet Order for 4-Word Read-Data Transfer
00	0 1 2 3
01	1 0 2 3
10	2 3 0 1
11	3 2 1 0

Interjecting Command Packets

Data transfers on the bus are either write-data packets issued with a write-command packet, or read-data packets. These transfers consist of multiple data beats. When a transfer contains multiple beats of data payload, a command packet can be interjected on an even-beat boundary. This feature allows new transactions to be started without having to wait for a long multi-beat data transfer to complete. This specification allows read-command packets and coherency-control packets to be interjected. Write, castout, push, and partial write operations, or other data packets cannot be interjected into a multiple-beat data transfer.

8.2.3 Transfer-Handshake Packets

The transfer-handshake bus is used to acknowledge command or data packets that were received on the inbound bus. Every command and data packet that is received on the inbound bus is acknowledged by a transfer-handshake packet on the associated outbound transfer-handshake bus. The transfer-handshake packet occurs a fixed number of beats later. Each transfer-handshake packet is two beats in length. *Table 8-11* shows the handshake encoding for the bus.

Table 8-11. Transfer-Handshake Definition

Response Beat 0, Beat 1	Description
0 0	Null/Idle
1 0	Acknowledge (command/data accepted)
0 1	Retry (command/data rejected, reissue command)
1 1	Parity error (parity error detected on bus)

The slave sends this acknowledgment packet to the bus *n* beats after receipt of the last beat of the command or data packet (*n* is the minimum number of beats necessary for the slave to receive the data from the bus, check the command and address, and generate the response). This time is implementation-dependent and may vary from one device to the next. The master samples the response STATLAT beats after the last beat of the command or data packet. STATLAT is the number of bus beats *between* the last beat of the command or data packet and the first beat of the acknowledgment packet. For example, if the last beat of a command packet was on beat *j* and the first beat of the acknowledgment packet occurred on beat *k*, then the value for STATLAT would be *k-j-1* (see *Figure 11-1* on page 370). The STATLAT beat count includes the time required by the slave to generate the response plus the time that it takes for the packet to be sent and the acknowledgment to be returned. For consistency in design of the processors that attach to this bus, an upper limit is defined for the time between the master issuing the last beat of the command or data packet to the bus to when it receives the first beat of the acknowledgment packet (see *Table 11-1* on page 371 and the *IBM PowerPC 970MP RISC Microprocessor Datasheet*). This time should be minimized to eliminate unnecessary delays on commands in the pipeline that have ordering requirements with the current command. The STATLAT parameter is configured by the inter-integrated circuit (I^2C) interface during the bus initialization phase.

8.2.3.1 Null Transfer Handshake

The null transfer handshake is the default response from a slave device. If the slave does not drive the transfer handshake with either an acknowledgment, retry, or parity error, then the response is, by default, null. The null response can occur due to a slave device timeout or a terminated transaction under certain, special conditions defined below.

Master:
For the master, this transfer-handshake response from the slave indicates that the command or data packet that the master sent was not accepted by the slave. Based on the address/data packet type, the master actions are as follows:

Command packet: The processor responds by going into checkstop.[1] If the command was a write command and the master detects this response before it has completed the full data transfer of the write-data packet, it can either complete the full data transfer or discard the remaining even-numbered data beats for the transfer.

Read-data packet: The processor responds by going into checkstop. A master always transmits full packets on the bus. The handshake is received after the end of the read-data packet (see *Figure 8-3* on page 233).

Note: The error might also result from a timeout while waiting for data or from an incorrect transfer size by the slave.

Write-data packet: The processor responds by going into checkstop, if the write-command packet associated with this packet received an acknowledgment transfer handshake from the slave. Otherwise, the null response is ignored. If the write command is retried by the slave, then the null response is the correct response for the data packet associated with that write command.

Slave:
The slave issues the null transfer handshake response for the following non-error conditional data packet for a write command that was retried. The command or data packet is discarded, and status is logged in the slave for the error case.

8.2.3.2 Transfer-Handshake Acknowledgment

The acknowledgment response indicates that the addressed slave accepted this command or data packet. If a bus agent[2] accepts (acknowledges) a command packet to send to a remote bus, it is responsible for completing the transaction back to the bus master if the remote bus does not accept the command packet. For a read transaction, this implies returning data to the master with the data error signal activated. The data error is signalled by asserting the 35th bit (DERR signal) of the even data beats. For writes, the command and data packets are discarded. The device must also have a mechanism to signal a machine check indicating that the error occurred.

Master:
For the master, the acknowledgment response indicates that the command or data packet was accepted and that it might complete execution of the packet transfer. Based on the packet type, the master acts as follows:

1. Hardware has detected a condition that it cannot resolve, and which prevents normal operation. It stops executing instructions, responding to interrupts, and so on.
2. Bus agents are devices such as the North Bridge, but not switches that might be used to relay command and data packets.

Command packet: For a command packet that results in data being returned by the slave, the acknowledgment response indicates that the command has been accepted and need not be reissued to the bus. Inbound data packets to complete the transaction may be received starting in the beat following the response. For a write-data packet, the acknowledgment response indicates that the command has been accepted. The slave cannot retry the data packet after it accepts the command. That is, an acknowledgment response for the command packet indicates that the slave has set aside buffer space for the write-data packet. For command packets, this response indicates that the command is complete.

Data packets: This response indicates to the master that the data being sent was accepted by the slave without errors.

Slave:
The slave issues this acknowledgment response when:

- The slave received the command packet with a valid transfer type, transfer size, and address.
- For write transactions, there is queue space for the command and data.

The slave stores command packets in a command queue and stores data packets in data buffers.

8.2.3.3 Transfer-Handshake Retry

A handshake retry may be issued to flow control the command packets when the slave does not have space for the command packet or the data packet associated with the command. Any command packet can be retried by the slave, except for reflected command packets. Data packets may not be retried.

Master:
For the master, the retry response indicates that the command was rejected by the slave for lack of space in the command queue or the data buffers. Based on the packet type, the master acts as follows:

Command packet: When the master receives a retry response for a command packet, it reissues the packet to the AD bus. If the command was a write command and the master detects this response before it has completed the full data transfer, then it can either complete the full data transfer, or discard the remaining even-numbered data beats for the transfer before reissuing the command packet.

Data packet: Retry responses are not valid for write-data packets and read-data packets.

Slave:
The slave issues this response when:

- The slave received the command packet but the command queue was full.
- If the packet was a write-command packet, there is no space for the command or the data.

To properly detect termination of a partial write-data packet, the slave must examine the Address Valid decode bits (see *Table 8-2 Command Packet Description* on page 237) on a per even-beat basis.

Note: The retry transfer handshake cannot be issued for write-data packets.

8.2.3.4 Transfer-Handshake Parity Error

This response is optionally issued whenever a single bit error is detected during any bus beat. It is an unrecoverable error that results in a machine check to the processor with all command and data packets in the pipeline being discarded.

Master:
For the master, the response is a hard error indicating that the bus is no longer functional. The processor responds by going into checkstop.

> **Command packet:** When the master receives a parity error response for a command packet, it reports the failure back to the system. The bus must be reinitialized before it can be used again.
>
> **Data packet:** Same as command packet errors.

Slave:
If the slave issues this response (optional), it should be within the normal packet response timings. (This packet error might make this timing determination imprecise.) For the slave, this condition is a hard error and the bus is no longer functional. The slave logs the error and reports it to the system. The error reporting mechanism is system-dependent.

8.3 Snoop Responses

Cache coherency is maintained using a global snoop method, where a memory controller device (the North Bridge) reflects command packets to all processors at the same time. Snooping is supported by dedicated snoop-response bus segments, consisting of one 2-bit SRO and one 2-bit SRI.

A snoop response begins when a processor receives a reflected command packet on the ADI bus. The processor starts a programmable timing chain that determines when the processor's SRO is driven and when the processor's SRI will be sampled.

The snoop response from each processor is transmitted on the SRO response bus in two beats (see *Table 8-12*). The North Bridge gathers the snoop responses from all processors and performs a logical OR operation on the accumulated responses. The North Bridge sends the logical OR of the snoop responses back to all processors on the SRI bus.

Table 8-12. Snoop-Response Bit Definition

Beat	Bits	Description
1	SR[0]	Intervention
1	SR[1]	Modified
2	SR[0]	Retry
2	SR[1]	Shared

Table 8-13. Allowed Snoop Responses

Retry	Intervention	Modified	Shared	Description
0	0	0	0	Null (exclusive for reads)
0	0	0	1	Shared
0	0	1	x	Modified
0	1	0	0	Invalid
0	1	0	1	Shared intervention
0	1	1	x	Modified intervention
1	x	x	x	Retry

8.3.1 Snoop-Response Bus Implementation

Each snoop-response bus is controlled by two configurable parameters: SNOOPLAT and SNOOPACC. For all parameters, time is measured in bus beats from the final locally clocked flip-flop or latch output to the first locally clocked input.

The processor SNOOPLAT parameter defines the number of bus beats between receiving the last beat of the reflected command packet and driving the first beat of the snoop response. SNOOPLAT does not need to be programmed for the processors, since the processors are assumed to be identical. The North Bridge SNOOPLAT value is the sum of the transfer time of the reflected command packet from the North Bridge to the processor, the processor SNOOPLAT value, and the transfer time of the snoop-response bus from the processor to the North Bridge (see *Figure 11-2 North Bridge Configurable Timing Parameters* on page 370).

On the North Bridge, the SNOOPACC parameter defines the delay between the time a processor sends the last beat of an individual snoop response to the time it receives the first beat of the accumulated snoop response from the North Bridge (see *Figure 11-3 Processor Configurable Timing Parameters* on page 371). SNOOPACC includes the time required by the North Bridge to gather the responses from all of the processors. The North Bridge reflects all incoming command packets at a pace determined by the SNOOPWIN parameter. SNOOPWIN sets the snoop window, which is the minimum distance between two consecutive snoop requests (see *Figure 11-3 Processor Configurable Timing Parameters* on page 371).

An address collision occurs if the current address is the same as a requested address for a previously received snoop. If this occurs, the current snoop request is delayed until the conflicting previous request is concluded. This condition is called previous adjacent address match (PAAM). The PAAMWIN parameter indicates the number of bus beats a request is active during which a conflicting snoop request cannot be issued. An unrelated snoop request can be sent during the PAAM window. *Figure 11-2 North Bridge Configurable Timing Parameters* on page 370 shows the timing of the PAAMWIN parameter.

For a snoop request to be issued, the following conditions must be satisfied:

1. At least SNOOPWIN beats have transpired since the previous snoop request was issued.
2. There is at least one non-active PAAM address slot available.
3. No active PAAM address conflicts with the request.

The number of PAAM address slots on the North Bridge is implementation-dependent, but ranges from two to four. A snoop request activates a PAAM address slot when it is issued. After PAAMWIN beats, the slot is deactivated and can be reassigned to another request. The number of address bits used to detect conflict is also implementation-dependent.

There is no requirement that all snoop requests fall in exact modulo SNOOPWIN beats. Even-numbered idle bus beats can be used beyond SNOOPWIN between two subsequent snoop requests. The PAAMWIN value is not required to be a multiple of the SNOOPWIN value.

The I^2C interface is used to program all programmable delay parameters (see *Section 11.1 I^2C Interface* on page 369).

8.3.2 Snoop-Response Descriptions

8.3.2.1 SResp Retry Response Code (Priority 1 - highest)

SResp Retry is issued for the following reasons:

- Lost reservation: A master that has a reservation will retry an atomic write/flush itself if the reservation has been lost since the write was issued.
- Push condition: A snooper will retry a transaction if a push is needed for a read or write-with-flush.
- Resource conflict: A snooper will retry a transaction due to collision with a resource that has ownership of the line.
- Memory and intervention buffer full: A North Bridge can retry a read transaction that might cause intervention, if it determines it temporarily cannot receive the intervention data. It is typically more efficient to use the transfer-handshake retry on the intervening data packet for this case.

SResp Retry ramifications:

- Master: May reissue this operation and use a different tag, or may reissue a different operation instead of or before this operation is reissued. Any data transfer aborted by this retry may be terminated prior to the data packet completion.
- Target: Any operation that has completed an SResp Retry may take a variable amount of time to clean up resources and, therefore, may cause future retries due to resources being tied up by this operation. Guarded cache-inhibited write operations need to be ordered with respect to each other. The processor cannot proceed and cannot issue the next operation until the SResp window with the null response has passed.
- Snooper: Any operation that has completed an SResp Retry is aborted by the snooper and leaves the cache state unmodified, except when Intervention is disabled on a read request and the snooper has modified data. The snooper will then push the data back to memory and clean or invalidate the line.

8.3.2.2 SResp Modified-Intervention Response Code (Priority 2)

The Modified coding is activated when a snooper detects the address of a cache line on a read operation that is contained in its own cache and is modified (dirty). The snooper then provides the data by using intervention.

SResp Modified-Intervention is asserted if a snooper asserts SResp Modified-Intervention on a Read or Read with Intent to Modify (RWITM) when bus intervention is enabled (N equals '1'), snooping is enabled (M equals '1') and a cache line is snooped modified. If SResp Retry is sampled instead of SResp Modified-Intervention, then the snooper can either push the block to memory or leave the cache state unmodified.

The ramifications of an SResp Modified - Intervention for bus devices are:

- Master and Read or RWITM:
  This tells the master that its request is satisfied by the cache holding the modified data.
- Memory and Read or RWITM:
  This tells the North Bridge to cancel its read request. If the command was read, the North Bridge looks for the tagged data and copies the block to memory.

8.3.2.3 SResp Shared-Intervention Response Code (Priority 3)

The Shared-Intervention coding is activated when a snooper detects the address of a cache line on a reflected read-command packet that is contained in the snooper's own cache and is the owner (most recent recipient) of the data. This signal can only be asserted by one bus device, since there is only one owner of data. Since SResp Retry is higher priority than SResp Shared, the snooper must wait until the snoop response is received before beginning the intervention push.

A snooper using this code must accommodate the option on burst reads whereby the requester indicates intervention is not wanted. In these cases, the response must be SResp Shared.

The ramifications of an SResp Shared-Intervention are:

- A master receiving this SResp code looks for intervention data.
- The North Bridge treats SResp Shared - Intervention as SRespRetry.

8.3.2.4 SResp Modified Response Code (Priority 4)

The Shared-Intervention coding is activated when a snooper detects the address of a cache line on a reflected read-command packet that is contained in the snooper's own cache and is the owner (most recent recipient) of the data. This signal can only be asserted by one bus device, since there is only one owner of data. Since SResp Retry is higher priority than SResp Shared, the snooper must wait until the snoop response is received before beginning the intervention push.

A snooper using this code must accommodate the option on burst reads whereby the requester indicates intervention is not wanted. In these cases, the response must be SResp Shared.

SResp Modified is asserted for the following reasons:

- A snooper asserts SResp Modified on a Read or RWITM when bus intervention is not enabled (N equals '0'), snooping is enabled (M equals '1'), and a cache line is snooped modified. If SResp Retry is sampled instead of SResp Modified, then the snooper can either push the block to memory or leave the cache state unmodified.
- A snooper asserts SResp Modified for flush or clean bus operations if the addressed block is modified. If SResp Modified is sampled in this case, the snooper pushes the block to memory and marks the cache Invalid (flush), or Shared/Exclusive (clean). If SResp Retry is sampled instead of SResp Modified, the snooper can either push the block to memory or leave the cache state unchanged.

8.3.2.5 SResp Shared Response Code (Priority 5)

Snooper:
The Shared response is encoded when a snooper inspects the address of a cache line on a read transaction that is contained in its own cache and has not been modified, marking the block shared if the block was marked exclusive. This signal can be asserted by more than one snooper, and the snooper will retain a copy of the block.

I/O Snooper:
I/O devices that do not cache data Exclusive or Modified (shared only) are allowed to assert without having the block cached (for example, they might snoop at a larger granularity than the block address).

Master:
This tells the bus master that the data on a read, when returned, must be marked shared and not exclusive.

8.3.2.6 SResp Null or Clean Response Code (Priority 6 - lowest)

The null or clean response is encoded to indicate one of the following:

- There is no local (or remote) device presently caching this line.
- A synchronize type of transaction (for example, **sync** or translation lookaside buffer sync [**tlbsync**]) has been completed by all snoopers.
- The line is cached, but the null response is allowed (for example, the null for a clean transaction that hits on an exclusive line).

8.4 Bus Transactions

This section provides details of the following processor interconnect bus transactions:

- Memory read transactions (general)
- Memory write transactions (general)
- Command only transactions

8.4.1 Terms

Each of the transactions in this section uses the following terms to define the parameters of the transaction:

Reservation	A reservation is an address location held by the processor. It is used to emulate atomic operations using the PowerPC load reserve indexed (**larx**) and store conditional (**stcx**) types of instructions. A processor has at most one reservation at any time. A reservation is established by executing a Load Word and Reserve Indexed (**lwarx**) or Load Double Word and Reserve Indexed (**ldarx**) instruction. It is normally lost when the corresponding Store Word Conditional Indexed (**stwcx.**) or Store Double Word Conditional Indexed (**stdcx.**) instruction is performed. A reservation might also be lost if the data at the address is modified by another processor or bus device.
Snooper	A bus device that inspects inbound reflected command packets and uses the snoop-response bus to keep cached data coherent with other system caches. A bus adapter or I/O bridge might contain a cache and, if so, will act like a snooper.
Memory	The bus device that responds to a memory read or write and handles positive acknowledgment for coherent operations. If some portion of memory is attached to a remote bus, the bus adapter also acts like memory for memory accesses to that remote memory space.
I/O Bridge	An I/O bridge device is a gateway to an I/O bus that cannot cache data in the Exclusive or Modified state. The bridge does not forward snoops to the I/O bus. If an I/O device has shared cache data, it is necessary to implement a directory for the cached data in the shared state.

8.4.2 Memory Read Transactions (General)

A master (processor) reads I/O or memory data by sending a read command to the memory controller of the North Bridge. The processor drives the ADO bus, provided it was not in the midst of sending another command packet, and there was no higher priority transaction ready to be sent. After a programmable number of beats (STATLAT), the master reads the transfer handshake from the THI bus to ascertain the status of the transfer. The slave (North Bridge) sends a positive acknowledgment on the THI bus if no parity error was detected and there was a slot to queue the read request. If no queuing space is available, a retry status is returned.

The North Bridge dequeues the request after internal arbitration and decodes the command packet. It issues a read request to the North Bridge for the indicated block size and reflects the command packet to all processors for snooping purposes. The North Bridge paces new snoop requests based on the programmable parameter, SNOOPWIN. The North Bridge will detect address collisions (transactions to the same cache line) and will delay the second conflicting transaction until PAAMWIN bus beats have transpired since the original conflicting transaction was issued for snooping. In addition, processors can request that transactions be handled one at a time, by setting the pipelined snoop (P) address modifier bit low.

Each processor drives their SRO bus during the snoop window that is seen by all processors and by the North Bridge at the same time. The processor may request that the transaction be retried with a retry snoop response. Otherwise, if a processor has a clean copy in its cache, the shared response code is returned. If the requested cache line is modified inside a processor cache, that processor signals the intervention snoop response, which is a promise to send to the North Bridge the modified copy in the form of a processor-to-memory read-data packet. The North Bridge accumulates the combined (logical-OR) snoop responses from all of the attached processors. Depending upon the combined response the North Bridge may abort, delay, or send the memory data or the intervened data to the original requester. The intervened data is also written to memory for regular read transactions (no intention to modify).

When the North Bridge responds with the read data, it sends a read-data packet, which consists of a 2-beat header and 2 to 32 beats of payload data. The header contains the original tag and the data size. The payload data is sent immediately after the header. The DERR bit is asserted if the data contains an error.

8.4.2.1 Read Transaction

A read command is issued to get data that is not immediately going to be modified. The modifier bits that are valid are N (intervention) and S (non-caching). The M and I modifiers are sourced from the page table entry, hardwired, or set by the I/O.

Master:
A read burst is issued by the processor to satisfy a load, tablewalk access, Data Cache Block Touch (**dcbt)** or other data prefetch, or instruction fetch (I-fetch) to a cacheable page that misses the cache. A read non-burst is caused by a non-cacheable load or I-fetch.

Atomic:
The Atomic modifier (TType [0]) is set along with the M bit when the read is to satisfy a **lwarx** or **ldarx**.

S Bit:
The S bit is set along with the M bit when the master will not cache the data but wants the latest copy. If S is set, a snooper is allowed to clean up dirty data in its cache by pushing it to memory, but keeping it marked exclusive afterwards.

N Bit:
The N bit is set when the master and memory are capable of intervention, intervention is wanted, and the read-data size is the coherency block size for the system. All non-block size reads must have the N bit set to zero. In addition, the processor is capable of setting N to '0' for all reads in case it is attached to memory that does not support intervention.

G Bit:
The G bit is set when the read is the result of a load to cache-inhibited guarded storage. When set, the system implementation knows this read might only complete once.

P Bit:
The P bit is set when snoop pipelining is allowed (default for reads). This bit can be cleared for reads if the processor requires the transaction snoop response to be resolved before another independent transaction is issued. When an address collision is detected, the North Bridge automatically delays the colliding transaction until the previous transaction is resolved.

Snooper:
If the address contained in the reflected command packet is in the cache and marked Modified, the snooper performs a push or intervention.

Memory:
Memory can provide the addressed data no earlier than the end of the snoop window for that transaction. The North Bridge examines the snoop-response bus, and, if it was SResp Retry or SResp Intervention, the North Bridge will terminate the operation and deallocate the tag. If the SResp response is Modified, because the North Bridge supports intervention, the North Bridge captures the line as it is transferred to the requester and stores the line to memory.

I/O Bridge:
If the G bit is set, an I/O bridge may not issue the read to any memory-mapped I/O devices more than once. This means waiting until the previous guarded read is committed (no retry from the transfer handshake) before sending the next request.

8.4.2.2 Read with No Intent to Cache Transaction

Read with no Intent to Cache (RWNITC) is another name for a read transaction with the S bit and M bit set (see above). It is a coherent read; that is, the master wants the latest data, but does not cache it. Therefore, the snooper can keep caching the data as Exclusive after it provides the data by a push or an intervention.

8.4.2.3 Read with Intent to Modify Burst Transaction

Master:
The RWITM transaction is issued by a master to bring an entire block into a cache for the purpose of writing to it. It is always a block-sized read. It is triggered by a store, **stwcx.**, **stdcx.**, or Data Cache Block Touch for Store (**dcbtst**) to a cacheable page that misses in the cache. The master should mark its cache Exclusive if the SResp is not Retry.

Snooper:
The snooper invalidates any line cached at the same physical block address and asserts SResp Null if marked Invalid, Shared, or Exclusive. If the request hits its cache, and it is marked Modified, the snooper performs either a push or an intervention. If the system supports Shared-Intervention and the request was marked with N set to '1', then the snooper can respond Shared-Intervention and push the data.

Memory:
The memory can provide the addressed data no earlier than after SNOOPACC. The North Bridge must examine the snoop-response code, and if it was Retry or Intervention, the North Bridge should terminate the operation and deallocate the tag.

Atomic:
The Atomic modifier (TType [0]) is set along with the M bit when the read is to satisfy a cacheable copy-back **stwcx.** or **stdcx.** The master SResp retries its own RWITM-A if the reservation is subsequently cleared after issuing the RWITM but before SResp and does not reissue the RWITM-A. If a processor does not support any cache levels below it (that is, it sees all the system coherency traffic), then the A bit need not be set on RWITM.

N Bit:
The N bit is set when the master and memory are capable of intervention, intervention is wanted, and the read-data size is the coherency block size for the system. All non-block size reads must have the N bit set to zero. In addition, the processor is capable of setting N to '0' for all reads, in case the memory does not support intervention.

G and S Bits:
These bits are not defined for RWITM.

8.4.2.4 LARX-Reserve Transaction

Master:
The LARX-Reserve transaction is an address-only transaction that sets the reservation for every cache level below the level serviced by a read atomic operation. If the reservation at one level is already set to the same address as the LARX or the LARX-Reserve being propagated, then it should not be propagated further because this causes a bus operation each time the LARX is executed and might be part of a program loop.

Snooper:
Does not see the LARX-Reserve for M equals '0'.

Memory:
Ignores this operation.

8.4.3 Memory Write Transactions (General)

A master sends a write command to write data to memory or to an I/O device. The write-command packet is immediately followed by a write-data packet. The slave (North Bridge) checks the command to see if there is buffer space to store the write-data packet. The slave responds with a retry transfer-handshake packet if there is insufficient buffer space. The master can then terminate sending the write-data packet on an even beat. It can then try again to send the write-command packet and write-data packet at a later time.

The North Bridge reflects every command packet to all processors. The snoopers ignore the reflected command packet if M equals '0'. Only the original processor needs to see the address inside the command packet to deallocate the tag after the transaction is completed. At that time, the North Bridge takes responsibility for snooping for the pushed (castout) data. The transaction must be propagated all the way to memory if the W bit is asserted.

8.4.3.1 Write-With-Kill Transaction

Master:
The write-with-kill transaction is a burst operation used to tell all snoopers to invalidate any copies of this line in their caches, while also storing the line to memory.

Table 8-14. Write-With-Kill Types Supported

WIM Bits for Write-With-Kill	W Bit	M Bit
Copy back due to load, store, or Data Cache Block Set to Zero (**dcbz**)	0	0
I/O Write[1]	0 or 1	1
Flush due to Data Cache Block Flush (**dcbf**)	1	0
Push due to snoop	1	0

1. An I/O write is a full cache line write from a memory address.

Snooper:
If M equals '0', the snooper ignores this operation. If M equals '1', the snooper treats this operation as a Data Line Kill (DKILL) to the same address block, marking it Invalid (this includes any store buffers) and the operation is passed to any higher level cache.

Memory:
Memory must not update storage if the transfer-handshake packet indicates Retry or the SResp value (if applicable) is Retry.

8.4.3.2 Write-With-Clean Transaction

Master:
A write-with-clean transaction is a burst operation caused by a processor executing a Data Cache Block Store (**dcbst**) instruction or a bus snoop read or clean to a modified block. It is used to tell all lower level caches that a copy still remains in this level, while updating memory (or I/O). Since no snooper has the line, it sets M to '0' so horizontal snooping is avoided. The block is written all the way to memory.

Snooper:
Snoopers should not see this operation since M equals '0'.

Memory:
Memory must not update storage if the SResp is Retry.

8.4.3.3 Write-With-Flush Transaction

Master:
A write-with-flush transaction is a partial-block write to memory and might be a sub-block burst operation from the I/O. It is used for cache-inhibited or write-through writes from a processor (sub-block writes). The processor sources the M bit from the page table entry. I/O masters can also use this for DMA writes to a cache block without getting ownership first. The processor will set M to '1' for this transaction.

Snooper:
If M equals '1' and the line is cached Modified, this operation is SResp Retried. The line is pushed back to memory with a write-with-kill, then invalidated (this includes any store back buffers [SBBs]). The only appropriate SResp response is Retry by a snooper (other than SResp Null, which is the default response).

Memory:
Memory must not update storage if the transfer handshake indicates retry or the SResp value (if applicable) is Retry. A bus agent cannot pass a write-with-flush to an I/O bus that might contain memory-mapped devices or memory that can be reserved without first successfully passing the response window on the processor interconnect bus.

8.4.4 Command-Only Transactions

8.4.4.1 DCLAIM Transaction (Invalidate Others)

Master:
A master issues a data line claim (DCLAIM) transaction to service a **dcbtst**, **dcbz**, or store instruction. The DCLAIM is used to attempt to take a coherent block from the shared (or, with **dcbz**, invalid) state to the modified state and all other horizontal caches to the invalid state. It differs from DKILL in that the DClaim does not invalidate the master's copy in a lower level (higher number) cache.

Snooper:
Snoopers must invalidate their data cache blocks if there is a hit on this address.

Memory:
Ignores this operation.

8.4.4.2 Flush Transaction

Master:
A flush transaction is caused by a **dcbf** that hits on a memory coherent cache block and is marked shared or invalid. It is sent to other snoopers that might have a copy of the line.

Snooper:
If M equals '1', snoopers snoop their caches, and, if the line is cached, it is marked invalid. If the line was marked as modified, it is pushed back to memory. A snooper might respond modified, or might respond Null. An SResp Retry response should only be used if the command cannot be accepted or a pipeline address collision occurs.

Memory:
Memory might ignore this operation, even if a snooper responds SResp Modified, since intervention is not supported on the flush operation itself. The flushed data is sent to memory on a separate write-with-kill operation.

8.4.4.3 Clean Transaction

Master:
A clean transaction is caused by a **dcbst** that hits on a memory coherent cache block and is marked shared or invalid. It is sent to other snoopers that might have a modified copy of the line.

Snooper:
If the line is cached, then it is marked shared (or exclusive if it is the lowest cache in the hierarchy). If it was marked modified, the line is pushed back to memory. A snooper might respond modified, or might respond Null. An SResp Retry response should only be used if the command cannot be accepted or a pipeline address collision occurs.

Memory:
Memory might ignore this operation, even if a snooper responds SResp Modified, since intervention is not supported on the clean operation itself. The cleaned data is sent to memory on a separate write-with-clean operation.

8.4.4.4 IKill Transaction

Master:
The intent of the Instruction Line Kill (IKILL) transaction is to invalidate entries in any instruction-only caches in the system. Data only or combined caches are invalidated with other coherency operations. An IKILL block is caused by an ICBI instruction that hits on an instruction cache block that is marked as memory coherent. In order to prevent bus livelocks, this command should be issued with the P bit set to '0'.

Snooper:
Snoopers must invalidate their instruction cache blocks if there is a hit on this address. Any unified data or data only cache does not need to be snooped. A snooper might respond Retry, or might respond Null. An SResp Retry response should only be used if the command cannot be accepted due to resource conflicts.

Memory:
Memory can ignore this operation.

8.4.4.5 TLBIE Transaction

Master:
TLBIE is caused by a processor executing a TLB Invalidate Entry (**tlbie**) instruction.

Snooper:
Snoopers accept this transaction regardless of the M bit and invalidate any TLBs in the congruence class.

Memory:
Memory can ignore this operation.

8.4.4.6 TLBSYNC Transaction

The intent of the TLBSYNC transaction is to act as a barrier that forces all previous operations using invalidated TLBs to complete before the TLBSYNC completes.

Master:
The master issues the TLBSYNC transaction in response to a processor **tlbsync** instruction.

Snooper:
Snoopers must SResp Retry the TLBSYNC until all previous loads or stores and I-fetches that used any TLBs have been flushed or performed and any snooped TLBIEs are completed. A snooped TLBSYNC has the same effect on a processor that a **sync** would have if it were executed on that processor.

Memory:
Memory can ignore this operation.

8.4.4.7 SYNC Transaction

Master
A master issues a SYNC when a processor executes a **sync** instruction. The master stops processing all future instructions until all previous instructions have been completed. Then the SYNC transaction is issued to the bus, and the **sync** instruction is not completed until the SYNC transaction completes on the bus. SResp Retry will cause the operation to be repeated; SResp Null signals completion. To prevent bus livelocks, this command should be issued with the P bit set to '0', if the snooper implementation would cause resource conflict retries.

Snooper:
A snooper drives SResp Retry if there are any snoop operations pending, or cache pushes or snoop operations pending from previously snooped bus operations. Otherwise, it responds SResp Null.

Memory:
Memory signals SResp Retry until stores are performed if they can be reordered within the memory unit. Otherwise, it responds SResp Null. Memory can also respond SResp Null. The SYNC is used as a store barrier.

8.4.4.8 EIEIO Transaction

Master:
The intent of the Enforce In-Order Execution of I/O (EIEIO) transaction is to act as a barrier for all non-cacheable loads or stores that follow it. It forces all previous non-cacheable operations to complete before any non-cacheable operation issued after the EIEIO. EIEIO is caused by a processor executing an **eieio** instruction.

Snooper or Memory:
Ignore this operation.

I/O Bridge or Bus Adapter:
Accept and propagate toward memory-mapped I/O storage and do not allow any cache-inhibited storage access to bypass (if they can be reordered).

8.4.4.9 Null Transaction

Master:
A Null transaction is used by the processor to break cyclic deadlocks or prescheduled transactions that are no longer needed.

Snooper, Bus Adapter, or Memory:
Ignore.

9. Power and Thermal Management

The power-management design of the 970MP microprocessor has two primary objectives: to achieve high performance whenever it is needed and to minimize the operating power during both active and idle periods. Power-management support includes frequency and voltage scaling, dynamic power management, and power down of one core while the second core continues to operate.

9.1 Definitions

9.1.1 Full Power Mode

Full Power (Full Run mode) is the default power mode of the processor. After initialization or reset, the processor will always be in this mode. All internal units are clocked at full clock speed and are fully operational.

9.1.2 Doze Mode

This mode is entered from Full Power mode after the processing core has been quiesced, and instruction fetch and data prefetch have ceased. This mode is a power saving mode, because only the circuitry needed to provide bus snooping capability and maintain memory coherency is active.

To enter Doze mode, set HID0[DOZE] to '1', and then set MSR[POW] to '1'. When Doze mode is entered this way, it will stay in this mode until interrupted out, rather than try to transition further into Nap mode. An interrupt condition such as an external interrupt, decrementer, hard reset (*hreset*), soft reset (*sreset*), or machine check is required to return to full power.

9.1.3 Nap Mode

Nap mode provides additional power savings beyond Doze mode. In general, clocks to all internal units are switched off. Only the timer/decrementer facility, the I/O circuitry, and part of the pervasive unit are clocked and operating. The phase-locked loop (PLL) is running and stays locked to the global system clock (SYSCLK). The clock mesh is operating, as is the bus clock.

To enter Nap mode, first the Nap bit in Hardware Implementation Dependent Register 0 (HID0[9]) must be set. Then the power-management bit in the Machine State Register (MSR[45]) must be set. The processor will then gate its core clocks and enter Doze mode. In Doze mode, the processor will continue to snoop. However, it asserts its quiescent request (QREQ) signal to indicate to the chip set that it is prepared to go into Nap mode if snooping is not required. If the chip set determines that no memory activity requires the processor to snoop, it asserts a quiescent acknowledgment (QACK). Once the processor detects the assertion of QACK, it transitions to Nap mode. While in Nap mode, QACK is monitored constantly. If it is dropped, the processor transitions back to Doze mode.

If the processor has to act upon an incoming snoop, the bus interface unit (BIU) becomes active, and QREQ is deasserted. However, the processor stays in Doze mode and waits for the BIU to become idle again. As soon as the BIU is idle, QREQ is issued again. QACK can be reactivated when the snoop is completed (after the snoop-response time). The processor switches back to Nap mode after QACK is received.

If QACK is received without QREQ being sent (for example, the BIU is not idle), the processor will enter an error state. If QACK is deactivated while the processor is switching to Nap mode, the transfer to Nap mode completes before the processor is brought back to Doze mode. Any external interrupt, reset, or check condition transfers from Nap mode back to Full Power mode.

9.1.4 Deep Nap Mode

In Deep Nap mode, the chip is in the same state as Nap, except that the clock frequency in the chip is reduced to 1/64th of the nominal frequency. If this state is enabled, it is entered immediately after entering Nap mode and exited immediately before exiting Nap mode.

9.1.5 Dynamic Power Management

Dynamic power management (DPM) refers to the cycle-by-cycle control of clocks, as hardware facilities are used for computation and then go idle for some cycles. This gating of clocks while circuits are idle saves power with no reduction in performance. On a cycle-by-cycle basis, DPM enables stopping a logical function based on the need for the function. DPM also enables stopping a pipeline stage in a unit based on a change in the content of the pipeline stage.

9.2 Power-Management Support

System software manages power dissipation in a number of ways, using a number of hardware facilities.

9.2.1 Power-Management Control Bits

Dynamic power management (DPM) refers to the cycle-by-cycle control of clocks as hardware facilities are used for computation, and then go idle for some cycles. This gating of clocks while circuits are idle saves power with no reduction in performance. In normal operation, DPM should be enabled. It can be disabled, however, by negating HID0[11]. To enter an idle state, software must first set a bit in HID0 to identify which idle mode is wanted, and then set MSR[45] to trigger the transition to that mode. Setting HID0[9] selects Nap mode; setting HID0[8] selects Doze mode; setting HID0[7] selects Deep Nap mode. *Table 9-1* summarizes these power-management control bits.

Table 9-1. Power-Management Control Bits

Bit	Bit Name	Power Saving Mode
HID0[7]	deep nap	Deep nap
HID0[8]	doze	Doze
HID0[9]	nap	Nap
HID0[11]	dpm	Dynamic power management enable
MSR[45]	POW	Power management enable
MSR[48]	EE	Enable exception (interrupt)

9.2.2 Interrupts

The only way to get from a power saving mode back into the Full Power mode is by asserting one of the following interrupts:

- External interrupt
- Decrementer interrupt

Before entering a power saving mode, the MSR[EE] bit must be set to enable these interrupts. After an interrupt is taken, the return from interrupt (**rfid**) or the hypervisor return from interrupt (**hrfid**) automatically resets the MSR[POW] bit, therefore, software must set it once again to reenter a power-saving mode.

9.2.3 Bus Snooping

The processor interconnect participates in the system power management through two asynchronous control signals called QREQ and QACK. QREQ is a processor output signal that is asynchronously sampled by the local clock of the North Bridge. QACK is a North Bridge output signal that is asynchronously sampled by the local clock of the processor and other bus masters.

Figure 9-1 on page 266 is a flowchart of the sequence of steps for the processor to enter Doze or Nap mode. *Figure 9-2* on page 267 is a flowchart of the sequence of complementary steps taken by the North Bridge in response to the assertion or negation of QREQ by a processor. In Doze mode, the processor must be capable of snooping all reflected command packets from the North Bridge. In Nap mode, the processor is not required to snoop transactions, although it must be capable of returning to Doze mode for the purpose of snooping if QACK is negated.

In the normal (or desired) sequence of events, the processor and North Bridge observe a 4-phase handshake for QREQ and QACK. The processor first asserts QREQ after the processor has quiesced, the snoopers are idle, and all outstanding processor interconnect bus transactions have completed. The processor then waits for the North Bridge to assert QACK. While the processor is waiting for the assertion of QACK, it is in an intermediate mode called Doze. Once the North Bridge asserts QACK, the processor enters Nap mode. To exit Nap mode, the processor negates QREQ and then waits for the North Bridge to negate QACK before returning to the Run state.

There are a few scenarios in which the 4-phase handshake is preempted.

1. While in Doze mode, the North Bridge reflects command packet snooping. The action taken by the processor is to negate QREQ while snooping the reflected command packet and while staying in Doze mode.
2. While in Doze mode, the processor receives an interrupt. The action taken by the processor is to negate QREQ and return to the Run state.
3. While in Nap mode, the North Bridge negates QACK while the processor has QREQ asserted. The processor must then return to Doze mode within 64 bus clocks so that it can return to snooping reflected command packets from the North Bridge.

As shown in *Figure 9-1* on page 266, the North Bridge normally negates QACK when QREQ is negated by any of the attached processors. However, it might also negate QACK if there is bus activity from any of the other attached bus devices that can be a bus master.

Figure 9-1. Processor QREQ/QACK Signalling

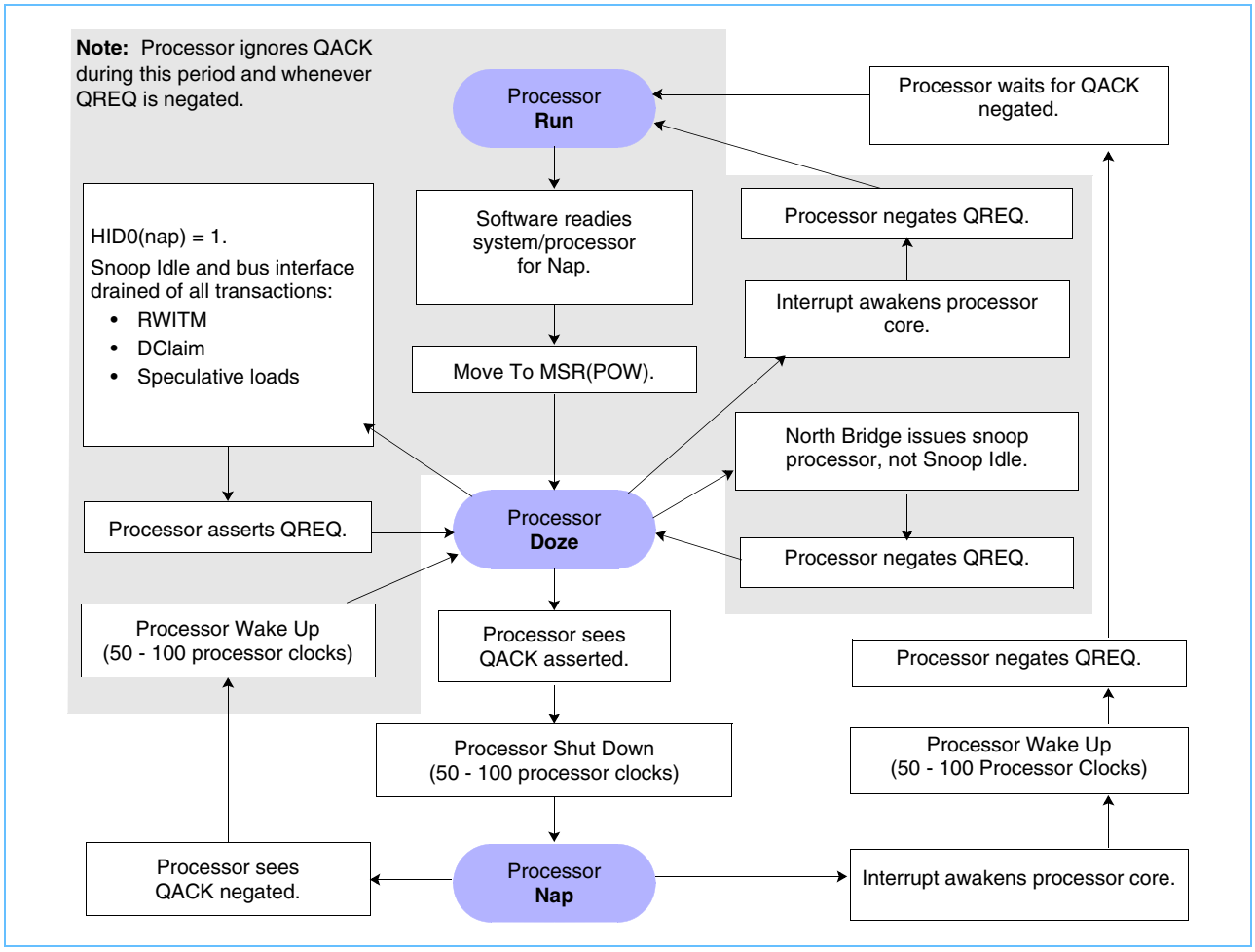
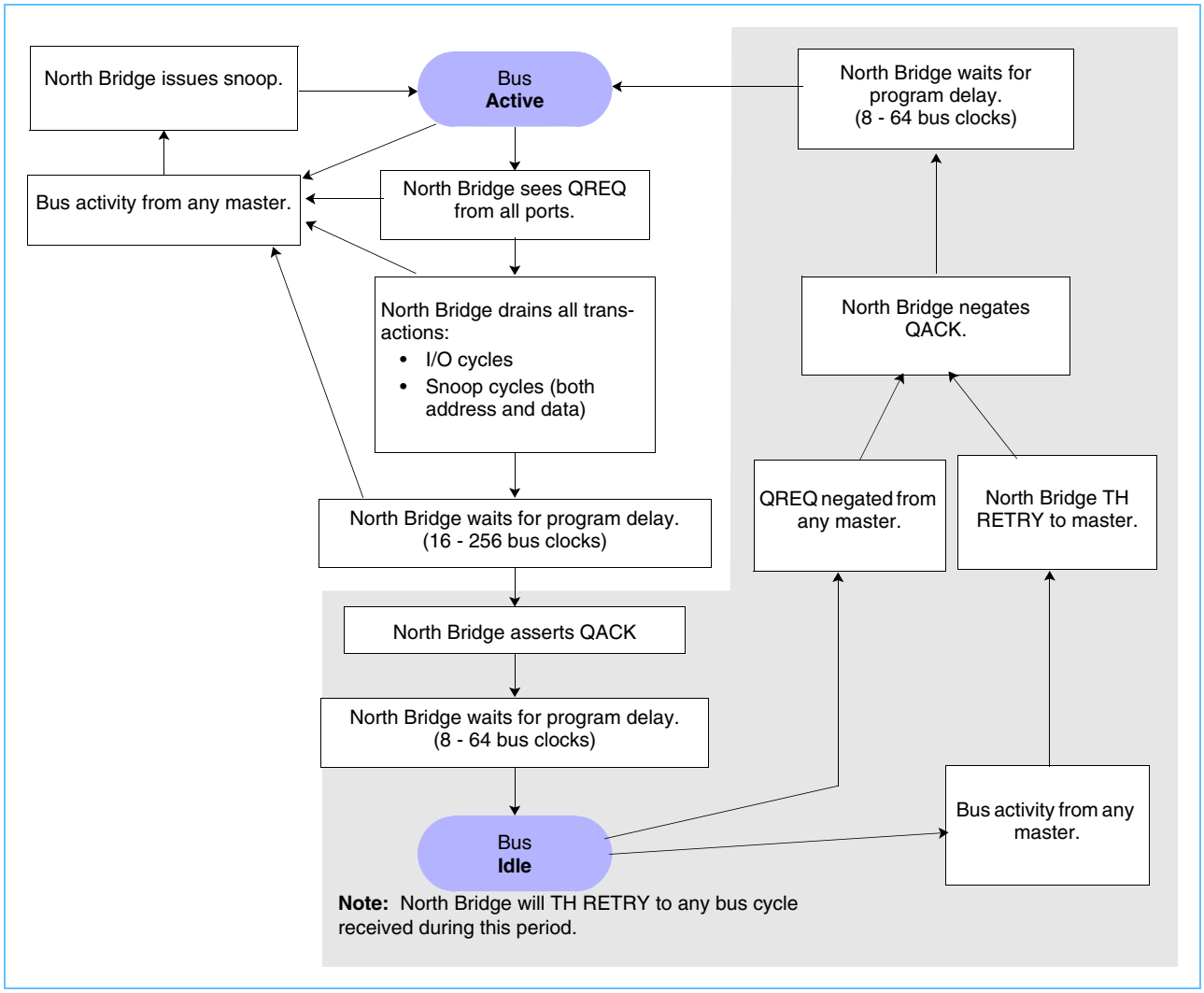


Figure 9-2. North Bridge QREQ/QACK Signalling

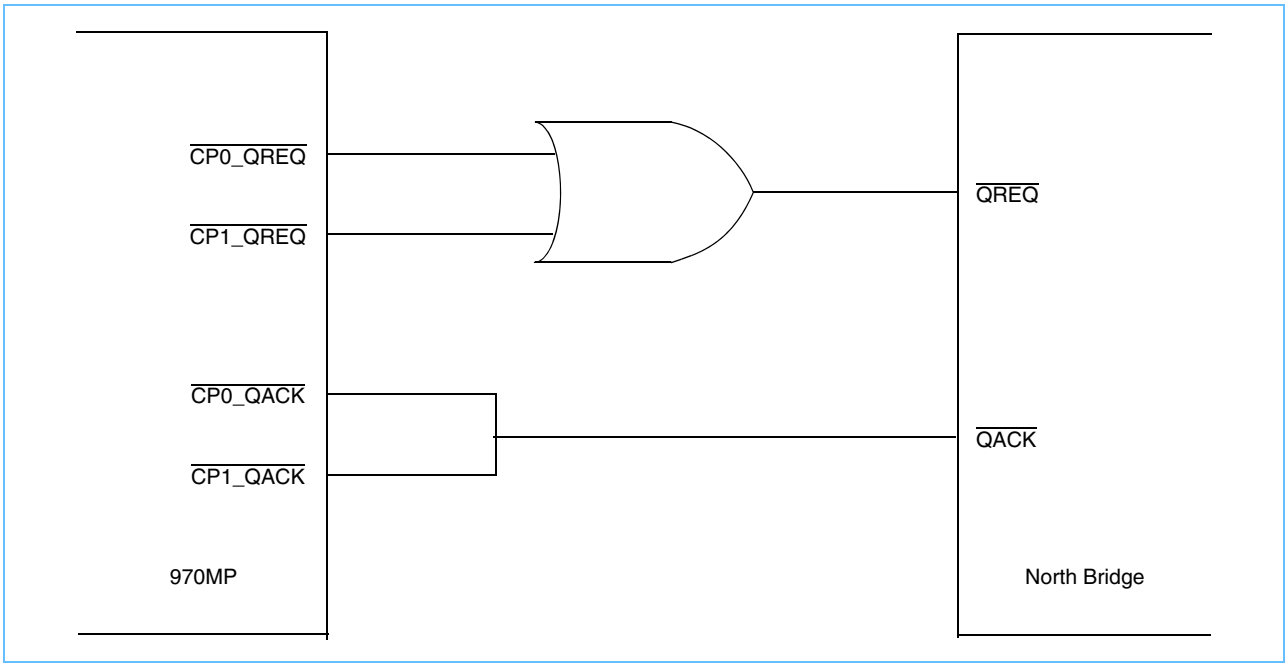


After the HID0[nap] bit is set and then the MSR[POW] bit is set, the processor enters Doze mode and asserts its QREQ signal. In this mode, core clocks are gated to reduce power, but clocks in the storage subsystem (STS) are still active to support bus snooping. The PLL, timers, and interrupt logic are also active in all the idle modes. The processor must remain in this Doze mode for as long as the system determines that snooping is required. The assertion of QREQ indicates to the system the processor's readiness to go into Nap mode. Once the system determines that snooping is not currently required, it can assert QACK. When the processor receives this signal, it completes the transition to Nap mode.

If snooping is required again, the system can negate QACK, signalling to the processor that it must transition back to Doze mode and begin snooping the bus. The general requirement is that the system must deassert QACK at least 64 bus cycles before it initiates bus activity to allow the processor to complete the transition back to Doze mode. However, the calculations below can be used to fine tune this delay. If this bus activity once again ceases, the system can assert QACK and the processor will go back into Nap mode.

The two pairs of QREQ/QACK signals in the 970MP design allow the two processors to independently Nap. It is possible to use the 970MP microprocessor with a North Bridge chip that supplies only a single QREQ/QACK pair. In this case, the two processors will be forced to go into and come out of Nap mode together. *Figure 9-3* shows the external logic and connectivity required to support the 970MP microprocessor with such a North Bridge.

Figure 9-3. Using a 970MP Microprocessor with a Single QREQ/QACK Pair



Because the QREQ signal is active low, an external OR gate is used to combine the two QREQ signals from the two processing units. Thus, QREQ is asserted to the North Bridge only if both processing units are asserting their QREQ signals. Once the North Bridge receives the asserted QREQ signal, its QACK signal is broadcast to both processing units by driving both QACK inputs from the single QACK on the North Bridge.

9.2.3.1 Delay Calculation

The requirement for the worst case QAckMinLowTime does not account for the case where the processor has not yet reached Nap or Deep Nap before QACK is negated. This additional delay can be accounted for by either increasing the required QAckIdleDelay or by imposing a requirement on QAckMinLowTime. Through *Table 9-2* and *Table 9-3*, below, we present the requirement as a minimum QAckIdleDelay, and a minimum combined QAckIdleDelay and QAckMinLowTime.

Table 9-2 provides the minimum QAckIdleDelay required for three different bus ratios and three different mesh clock frequencies. The required delay is 24 full frequency (f) processor clocks plus 195 mesh frequency (f, f/2 or f/4) processor clocks. Since the bus clock frequency scales with the mesh clock, the relation between these is a function of the bus ratio, but independent of frequency scaling. At 2:1, there are 4 bus clocks per mesh clock, at 3:1 there are 6 bus clocks per mesh clock, and at 4:1 there are 8 bus clocks per mesh clock. Since the 24 full frequency clocks do not scale with the mesh and bus clocks, the relation between these full frequency clocks and the bus clocks depends on both the bus ratio and the scaled frequency. At 2:1, there are 4 bus clocks per full frequency clock at f, 8 at f/2, and 16 at f/4. At 3:1, these values are 6 bu clocks per full frequency clock at f, 12 at f/2, and 24 at f/4. At 4:1, these values are 8 at f, 16 at f/2, and 32 at f/4.

Table 9-2. Minimum QAckIdleDelay requirement in bus clocks for 970MP

	2:1	3:1	4:1
f	55	37	28
f/2	52	35	26
f/4	51	34	26

Table 9-3 provides the minimum (QAckIdleDelay + QAckMinLowTime) required for three different bus ratios and three different mesh clock frequencies. The required delay is 48 full frequency (f) processor clocks plus 309 mesh frequency (f, f/2, or f/4) processor clocks.

Table 9-3. Minimum (QAckIdleDelay + QAckMinLowTime) requirement in bus clocks for 970MP

	2:1	3:1	4:1
f	90	60	45
f/2	84	56	42
f/4	81	42	41

Note: The default values for these two parameters are QAckIdleDelay = 50, and QAckMinLowTime = 6. This combination satisfies both of the requirements in the two tables above for 3:1 mode at f/2, and so this combination is appropriate for use at this bus ratio and frequency configuration.

9.2.4 Thermal Diodes

Thermal diodes are placed near the hot spot on each core and brought off chip separately. External logic can be used to monitor the diode temperature of the two cores independently for managing power based on thermal limits.

9.2.5 Bus States while in Power Saving Modes

When serving snoops, the BIU is active and drives the outputs as required. When in Nap mode, there is no snooping.

- Data Out Bus, Transfer Handshake Out, and Coherence Response are driven to an idle mode.
- Clock Out is always driven with the proper clock signal.
- Clock In expects to receive a clock signal.

9.3 Software Considerations for Power Management

9.3.1 Entering Power Saving Mode

The following code sequence should be used to enter a power save mode

```
.......
.......
mthid0 (NAP)
.......
.......
.......
.......
loop:  dssall (VPU prefetching stop)
sync
mtmsr (POW)
isync
br    loop
.......
.......
```

The Data Stream Stop All (**dssall**) instruction is needed to stop those prefetch engines started in behalf of the vector processing unit (VPU) prefetches. Only the above sequence will bring the processor into the power save mode. Switching the Move To HID0 (**mthid0**) instruction and the Move To Machine Status Register (**mtmsr**) instruction in the above sequence does not result in a switch to a power saving mode. When an interrupt is taken, it resets the MSR[POW] bit.

9.3.2 External Interrupt Enable

Only an external interrupt or timer interrupt will bring the processor back from the power save mode. Therefore, note that MSR[EE] must be set before entering the above loop. Failing to set MSR[EE] and applying an external interrupt or a timer interrupt will result in unpredictable behavior by the processor.

9.4 Power Tuning Facility Overview

The power tuning facility is the heart of the power management for the 970MP microprocessor. It controls the power-management modes, on-chip and off-chip clock frequency, and supports voltage scaling for the 970MP microprocessor. *Table 9-4* lists the Power tuning modes supported by the 970MP microprocessor.

Table 9-4. Power-Management Modes

Static Power-Management Modes	Frequency Scaling
Full, Doze, Nap	f
Full, Doze, Nap	f/2
Full, Doze, Nap	f/4
Deep Nap	f/64

Note:

1. See the *IBM PowerPC 970MP RISC Microprocessor Datasheet* for actual power dissipation specifications.

In the system, all processing units and the processing unit interfaces in the North Bridge change the power tuning mode (except for Deep Nap mode) concurrently. Any processing unit can request the mode change. This information is then transmitted to the North Bridge through the processor interface bus as a special request. The North Bridge grants the requests and mirrors this special request to all processing units. The North Bridge then waits for all processing units to signal that they have quiesced the bus and are ready to switch modes. The North Bridge then triggers the mode switch. It completes within 200 ns for bus ratios of 2:1, 3:1, 4:1, 6:1; and within 300 ns for a bus ratio of 12:1.

Frequency scaling on the processor interface bus requires changing the RoundTripDelay and TargetTime parameters in all the processing units and in the North Bridge. Because the I/O voltage is not changed, an initial alignment pattern (IAP) procedure is not required. The new parameters are sent along with the power tuning command; they overwrite the old parameters when the frequency switch occurs. No parameter change is required for the deep nap frequency, because there is no bus activity in this mode.

When switching power tuning modes, consideration must be given to the following items:

- Switching from high to low frequency will result in loss of accuracy and resolution in the decrementer counter and will slow reaction on interrupts. The operating system has to set the decrementer counter in order to prevent event and interrupt misses or queue overflow on external devices.
- Some interfaces must be running at a constant speed and voltage independent of the internal frequency and voltage (for example, the inter-integrated circuit [I^2C] interface, SDRAM interface, and PCI interface).

9.4.1 Power Tuning Facility Definitions

Bus clock (Bclk) — The external bus clock has half the frequency of the data clock due to the double data rate transmission mode on the processor interconnect.

Data clock (Dclk) — The bus data clock has a frequency of 1/*n*th of the mesh clock, where *n* is the bus ratio. Valid values for the bus ratio are 2, 3, 4, 6, and 12; with 8 and 16 supported only for test purposes.

Local clock (lclk) — The full frequency clock as delivered by the PLL, but with the same analog delay as the mesh. Every rising edge on the mesh clock has a concurrent rising edge on lclk with a small skew.

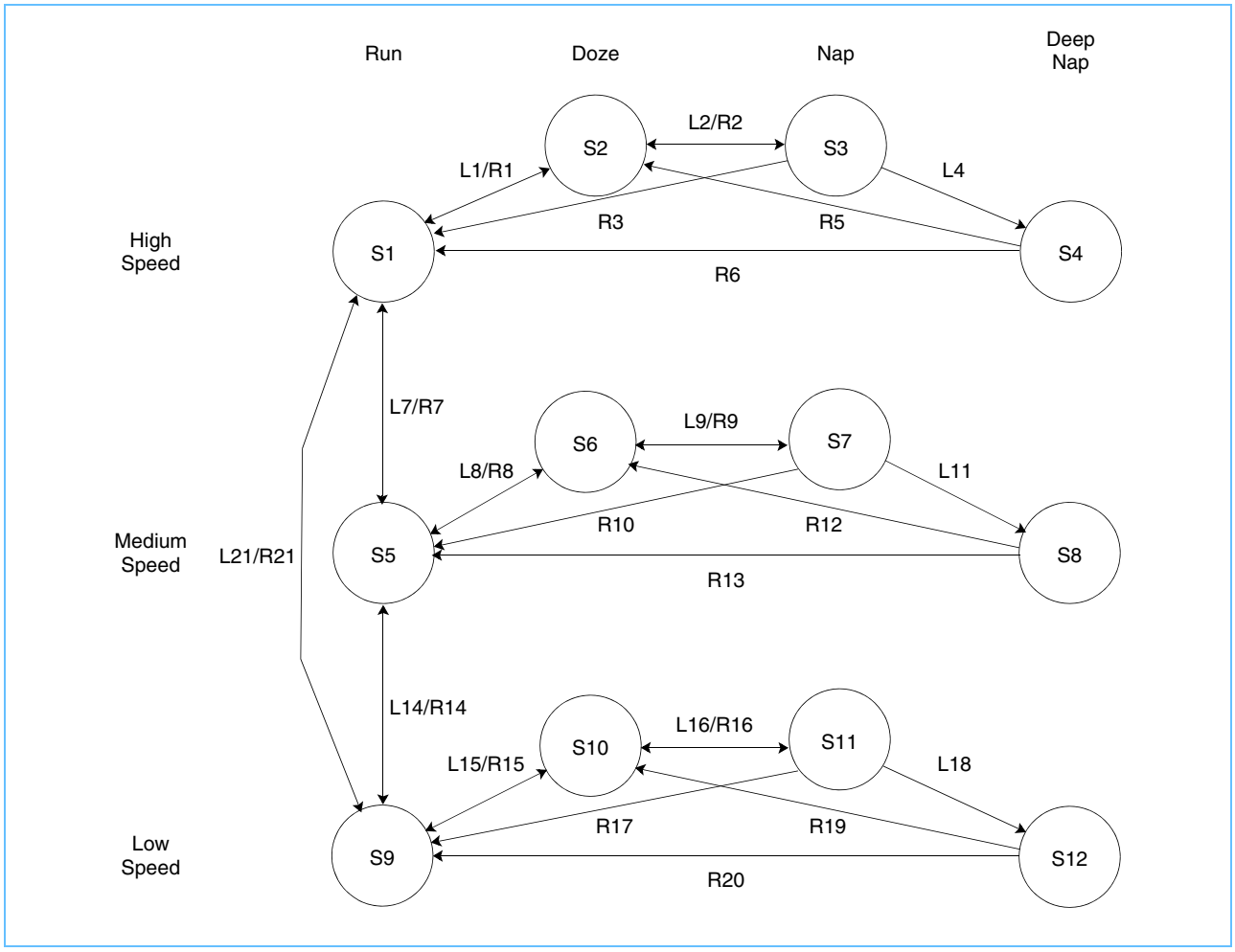
Mesh clock (mclk)	The logic behind the PLL generates full, half, or quarter frequency of the PLL clock and sends it on the mesh. The PLL also guarantees that some rising edge of the mesh clock at a latch is aligned to some rising edge of the SYSCLK when using full frequency.
PI	Processor interconnect bus (processor interface).
PLL/full frequency clock	Frequency is either 8 or 12 times the SYSCLK.
psync	A signal provided by the North Bridge that is active for one rising edge of SYSCLK every 24 SYSCLK cycles.
psync edge	A special mesh clock rising edge, aligned with a rising edge of the SYSCLK while the external *psync* is active.
SYSCLK	This is the system clock as provided on the board.
Time0	Time0 mark. A special rising edge on the bus clock, which is either concurrent to the *psync* edge or removed by two external bus clock phases (4 * x bus clocks edges). **Note:** Not all bus ratios are valid for all frequency modes (full, half, and quarter), considering that the external *psync* is a 24:1 SYSCLK signal, and taking the PLL multiplication factors and the Time0 definition into account.

9.4.2 Power Modes

Figure 9-4 is a state diagram showing the various power modes supported in the 970MP microprocessor and the transitions between them. *Table 9-5* on page 274 identifies the power states in that diagram, and *Table 9-6* on page 275 identifies the transitions labeled in the diagram.

Note: While the frequency associated with Deep Nap is 1/64 of full frequency in all cases, the three Deep Nap states are distinguished by the frequency of the processor that they transition from and to. When the processor goes into Deep Nap from full frequency, it will return to full frequency when it leaves Deep Nap. Similarly, it returns to half frequency if it came from half frequency or returns to quarter frequency if it came from quarter frequency. In order for these transitions between Run and Deep Nap to be fast, the voltage applied to the processor in Deep Nap will be the frequency required by the state it will return to. The power dissipation associated with the three different Deep Nap states will therefore not be the same, in general.

Figure 9-4. 970MP Power Mode States



There are 12 transitions that *lower* power dissipation, indicated as L*x*, corresponding to left-to-right or top-to-bottom transitions in the diagram. Corresponding to nine of these is a reciprocal transition that *raises* power dissipation, indicated as R*x*, and corresponding to the right-to-left or bottom-to-top direction in the diagram. In addition, there are three lower (L4, L11, and L18) and nine raise transitions (R3, R5, R6, R10, R12, R13, R17, R19 and R20) that do not have corresponding reciprocal transitions.

Table 9-5 describes the 12 power mode states.

Table 9-5. Power Mode States

State	Description
S1	Full Run, high speed
S2	Doze, high speed
S3	Nap, high speed
S4	Deep Nap, high speed
S5	Full Run, medium speed
S6	Doze, medium speed
S7	Nap, medium speed
S8	Deep Nap, medium speed
S9	Full Run, low speed
S10	Doze, low speed
S11	Nap, low speed
S12	Deep Nap, low speed

The three Full Run modes (S1, S5, S9), one each for high, medium, and low speed, correspond to all operating processor functions. The three Doze modes (S2, S6, S10) involve limited functionality, which include bus snooping, but not instruction execution. The timers, both decrementer and time base, continue to run during Doze modes, as does the logic for responding to interrupts. The three Nap modes (S3, S7, S11) correspond to a level of functionality below Doze, in which snooping is not supported. However, timer and interrupt logic are still active. The Deep Nap modes (S4, S8, S12) corresponds to the same functionality as Nap mode, but with the clocks running at 1/64 of full speed.

The state transitions between Run, Doze, and Nap at a given frequency are triggered as in the PowerPC 970MP microprocessor. Transitions associated with scaling the power tuning frequency are L7, R7, L14, R14, L21, and R21. These are triggered by the execution of a power tuning command, which is initiated by a write to the Power Control Register. These transitions are all summarized in *Table 9-6* on page 275 and briefly described below.

Table 9-6. Transitions between Power Modes

Transition	From	To	Trigger
L1	Run, High	Doze, High	MSR[POW] with HID0[nap] = '1'
R1	Doze, High	Run, High	Interrupt
L2	Doze, High	Nap, High	QACK asserted
R2	Nap, High	Doze, High	QACK negated
R3	Nap, High	Run, High	Interrupt
L4	Nap, High	Deep Nap, High	HID0[deep nap] = '1'
R5	Deep Nap, High	Doze, High	QACK negated
R6	Deep Nap, High	Run, High	Interrupt
L7	Run, High	Run, Medium	Power tuning command
R7	Run, Medium	Run, High	Power tuning command
L8	Run, Medium	Doze, Medium	MSR[POW] with HID0[nap] equal to '1'
R8	Doze, Medium	Run, Medium	Interrupt
L9	Doze, Medium	Nap, Medium	QACK asserted
R9	Nap, Medium	Doze, Medium	QACK negated
R10	Nap, Medium	Run, Medium	Interrupt
L11	Nap, Medium	Deep Nap, Medium	HID0[deep nap] equal to '1'
R12	Deep Nap, Medium	Doze, Medium	QACK negated
R13	Deep Nap, Medium	Run, Medium	Interrupt
L14	Run, Medium	Run, Low	Power tuning command
R14	Run, Low	Run, Medium	Power tuning command
L15	Run, Low	Doze, Low	MSR[POW] with HID0[nap] equal to '1'
R15	Doze, Low	Run, Low	Interrupt
L16	Doze, Low	Nap, Low	QACK asserted
R16	Nap, Low	Doze, Low	QACK negated
R17	Nap, Low	Run, Low	Interrupt
L18	Nap, Low	Deep Nap	HID0[deep nap] equal to '1'
R19	Deep Nap	Doze, Low	QACK negated
R20	Deep Nap	Run, Low	Interrupt
L21	Run, High	Run, Low	Power tuning command
R21	Run, Low	Run, High	Power tuning command

Software initiates the transition from a Full Run mode to a corresponding Doze mode by setting the MSR[POW] bit to a '1', when the HID0[nap] bit is a '1'. This triggers the normal idle mode sequence:

- Instruction fetch quiesces.
- The BIU quiesces.
- The clocks to the core are gated.
- QREQ is asserted.

At this point, the processor is in Doze mode. If or when QACK is asserted, the clocks driving the snoop logic are gated, and the processor enters Nap mode. Once in Nap mode, if QACK is negated, the snoop logic is reactivated and the processor returns to Doze mode. From either Doze mode or Nap mode, an interrupt

(External, Decrementer, System Management, Performance Monitor, or Reset) will reactivate all the clocks. This returns the processor to Full Run mode, where it will execute instructions starting at the corresponding interrupt vector. This brief description of the transitions among corresponding Full Run, Doze, and Nap modes applies to all three processing speeds (high, medium, and low).

The transition to Deep Nap (L4, L11, and L18) occurs immediately after the transition to Nap, if the Deep Nap enable bit (HID0[deep nap]) is set. In this state, the processor frequency is lowered to 1/64 of its full-speed frequency. Otherwise, the processor behavior is the same as in Nap state. In particular, if QACK is negated during Deep Nap, the processor transitions into Doze mode. If an interrupt occurs while in Deep Nap mode, the processor transitions to Run mode at the previous frequency.

Under normal operation, in which both cores are powered and participating (or prepared to participate) in program execution, system software controls the power mode of each processing unit. *Table 9-7* lists valid combinations of power modes for the two processors.

Table 9-7. Valid Combinations of Power Modes

Processor 0	Processor 1
Run	Run
Run	Doze
Doze	Run
Doze	Doze
Nap	Nap
Deep Nap	Deep Nap

In all these cases, both processors are clocked at the same frequency (there is a single PLL on chip) and are powered at the same voltage. *Table 9-7* shows that both processing units can be in the same power mode, or one can be dozing while the other is running. Because the two processing units have separate QREQ and QACK signals, it is also possible for one processor to nap while the other is running (or dozing). However, this would require the napping processor to first flush its L2 cache so that it would no longer have to snoop the bus.

9.4.3 Power Transition Latencies

Each of the transitions in *Table 9-6* on page 275 has a change in power dissipation associated with it, as well as a latency for the state change. The 970MP design incorporates several mechanisms to control these transition latencies in order to reduce the induced voltage spike that would otherwise occur.

There are three situations in which the power requirements of the processor can change drastically. One occurs during Run mode, when the instruction stream being executed changes from a low activity application to a high activity application. Because the hardware cannot detect this case, the system must be designed with sufficient decoupling capacitance to control the rate of current change (di/dt) associated with this type of power transition.

The second situation is that in which the processor transitions between Run and idle (Doze, Nap, Deep Nap) modes, at a given voltage and at constant (or Deep Nap) frequency. For this case, the 970MP microprocessor implements a programmable delay, which is inserted at several points in the transition sequence when coming out of idle modes back to the Run mode. This facility is described in detail in *Section 9.4.3.1*.

The third type of power transition is that associated with frequency changes in the power tuning facility. In this case, the dithering mechanism introduced in the 970FX microprocessor is expanded to handle the higher frequency design points of the 970MP microprocessor, as well as the quarter to full frequency transition in the power tuning facility. Clock dithering in the 970MP microprocessor is described in *Section 9.8.3 Clock Dithering* on page 290.

9.4.3.1 Idle to Run Transitions

In order to reduce di/dt, the transition from Deep Nap to Run mode is partitioned into several phases, with a programmable delay incorporated within phases. In the Clock Ramping Configuration Register, a 6-bit value (ranging from 0 to 63) specifies the programmable delay. It specifies the number of cycles the processor will spend at six different stages in the transition from Deep Nap to Run mode. To make these delays nearly equal for full, half, and quarter frequency transitions, the following approach is used:

- The full six bits are used to specify the number of full frequency delay cycles.
- The high-order five bits are used to specify the number of half-frequency delay cycles.
- The high-order four bits are used to specify the number of quarter-frequency delay cycles.

For example, a value of 12 placed in the register would result in 12 full frequency, 6 half frequency or 3 quarter frequency cycles of delay. All three correspond to the same 4 ns delay on a 3.0 GHz processor. A value of 27 placed in the register would result in 27 full frequency, 13 half frequency, or 6 quarter frequency cycles of delay. These correspond to 9 ns (full frequency), 8.7 ns (half frequency), and 8 ns(quarter frequency) delays on a 3.0 GHz processor.

Note: The delay is specified in core clocks, so the absolute delay stays constant when scaling the frequency.

Table 9-8 on page 278 provides the number of full frequency clock cycles of latency in the four phases in the Deep-Nap-to-Run mode transition for each of the three mesh frequency settings.

Note that the transition from Deep-Nap-to-Run passes through the Nap and Doze states as power is gradually increased to support Run mode. The four phases are:

- Phase 1: Interrupt during Deep Nap to Nap
- Phase 2: Nap to Doze
- Phase 3: Doze to Run
- Phase 4: Interrupt presented to running processor

The parameters C_f, C_h, and C_q represent the number of full frequency cycles in the programmable delay when in full, half, and quarter frequency, respectively. These delays occur in both the Nap-to-Doze and the Doze-to-Run phases of the transition, just after (i) C2[1] clocks are issued every other cycle, (ii) C2 clocks are fully enabled, and (iii) C1[2] clocks are (fully) enabled.

1. C2 is the clock for the slave latch.
2. C1 is the clock for the master latch.

Table 9-8. Latency of Deep-Nap-to-Run Transitions in Full Frequency Cycles

	Full Frequency	Half Frequency	Quarter Frequency
Phase 1	44	70	122
Phase 2	$40 + 3C_f$	$80 + 3C_h$	$160 + 3C_q$
Phase 3	$24 + 3C_f$	$48 + 3C_h$	$96 + 3C_q$
Phase 4	15	30	60
Total	$123 + 6C_f$	$228 + 6C_h$	$438 + 6C_q$

For example, if the 6-bit programmable delay value is set to 12, then $C_f = C_h = C_q = 12$. The latency for the Deep-Nap-to-Run transition for full, half, or quarter frequency would be 195, 300, or 510 full frequency cycles, corresponding to 65, 100, or 170 ns at 3.0 GHz, respectively. For a programmable delay value of 27, $C_f = 27$, $C_h = 26$, and $C_q = 24$. This yields latencies for full, half, or quarter frequency of 285, 384 or 582 full frequency cycles, corresponding to 95, 128 or 194 ns at 3.0 GHz, respectively.

9.4.3.2 Exiting Deep Nap Using a Decrementer Interrupt

The timer registers (Time Base and Decrementer) are updated at a rate that depends on the mesh frequency. When the mesh frequency is sufficiently low, the timers are adjusted by more than one tick on each update in order to maintain accuracy. At such a frequency, the precision of the timers is reduced. In particular, during Deep Nap the mesh frequency will fall below that required to maintain the precision of the timers set by the time-base enable (*tben*) frequency (or the PLL frequency, in the case of internally clocked timers). The resulting loss of precision has no effect on the accuracy of the timers. It also has no visible effect on the time-base precision, because the Time-Base Register is not accessed during Deep Nap. This loss of precision can affect the latency of the processor in detecting the interrupt signal when the decrementer value goes negative.

However, the incremental latency due to this loss of precision in the decrementer is quite low in the 970MP microprocessor. To reduce the latency associated with exiting Deep Nap due to a decrementer interrupt, the timers are updated once every mesh cycle on the 970MP microprocessor. The additional latency for exiting Deep Nap due to the lower precision of the decrementer is the mesh clock period minus the target timer period. For example, if the *tben* input is driven at 66 MHz to clock the timers, and the full frequency processor clock is running at 1.5 GHz, the added latency is computed as follows. The mesh clock frequency in Deep Nap is 1.5 GHz / 64 or 23.4 MHz, so the mesh clock period is 43 ns. The period of the *tben* input is 15 ns. So, the added latency to exit Deep Nap due to a decrementer interrupt is 43 - 15 = 28 ns. For higher frequency processors, this latency is less.

9.4.3.3 Frequency Transitions in the Power Tuning Facility

Transitions in the power tuning facility involve changing the mesh frequency while in Run mode. The frequency is changed from frequency one (F1), either full, half or quarter mode, to frequency two (F2), either full, half or quarter mode (but not the same as frequency one). These transitions are initiated by software changing the Power Control Register (PCR) in one of the processors, which causes a special bus transaction to the North Bridge. This transaction is reflected by the North Bridge to all processors in the system, which causes those processors to begin the transition process. Processors indicate their readiness to make the frequency switch itself by asserting QREQ to the North Bridge, and the North Bridge responds when it is ready and after it has received all the QREQs from the processors by asserting QACK to all processors. The

time it takes to get to this point in the procedure varies. It depends on activity levels in the processors and North Bridge. A typical implementation might achieve a 200 ns to 300 ns latency for this sequence when transitioning from full to half frequency on a 2.0 GHz processor.

From the time the processor receives QACK, it takes 20 additional F1 mesh clocks, plus eight full frequency clocks, to prepare for the frequency change. The frequency change itself occurs over the course of 24 or 48 cycles (selectable by a mode bit—the previous design supported only a 24-cycle dither). The frequency is dithered between F1 and F2, so that the frequency changes gradually from one to the other. These 24 or 48 cycles are always at the lower of F1 and F2. Once the frequency switch is finished, it takes 49 additional F2 mesh clocks (assuming a 3:1 bus) before the processor negates QREQ, signaling the end of the transition.

In the case of the full-to-quarter or quarter-to-full transition, the dithering described above is between F1 and half frequency. After the 32-cycle pause at half frequency, a second dithering sequence between half frequency and F2 takes place. The rest of these transitions are the same as described above.

The latency of the transition of the power tuning facility from QACK assertion to QREQ negation in cycles is:

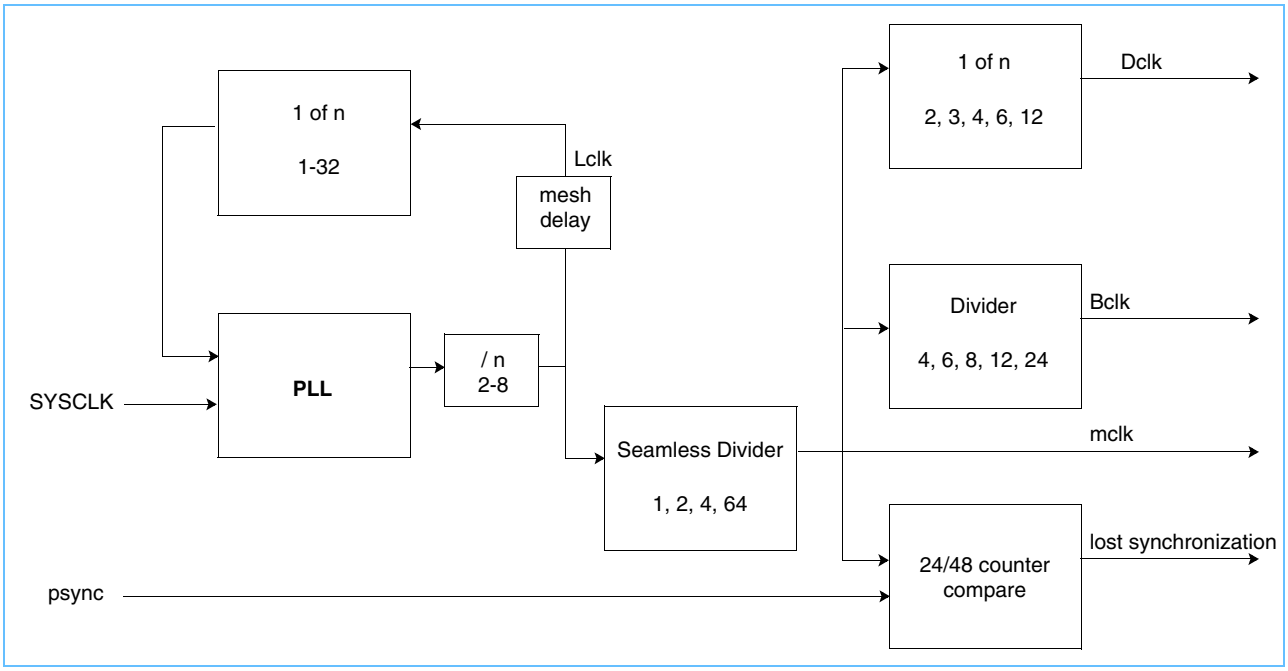
- 8 full + 20 F1 + 24 minimum (F1, F2) + 49 F2
- Plus an additional 24 minimum (F1, F2) when using a 48-cycle dither
- Plus an additional (32 + 24) F_{half} when executing full-to-quarter or quarter-to-full

9.5 PLL Design

The PLL is designed to support the frequency scaling capability of the 970MP microprocessor. A single PLL drives the clock mesh, with the circuitry of the power tuning facility from PU0 controlling the mesh frequency. Both the processor clock and the bus clock are derived from the reference clock input to the chip in the 970MP design. For frequency scaling, it is assumed that the reference clock, SYSCLK, and the related synchronizing clock, *psync*, run at a constant frequency.

The PLL uses a fixed divider in the feedback path, but a variable, seamlessly switched divider in the forward path. The fixed feedback path allows the PLL to constantly run at a fixed frequency, avoiding the need to relock when switching frequencies. The processor clock (mclk) and bus clock (Bclk) frequencies can be changed seamlessly, while maintaining the ratio between these two clocks at a fixed value. *Figure 9-5* on page 280 shows this design. Note that the processor interface supports a double data rate bus. Therefore, the data rate clock (Dclk) is twice the Bclk frequency, and is constrained by the processor design to be no more than half the mclk frequency.

Figure 9-5. PLL Design



The PLL is designed to allow a feedback divider value ranging from 1 to 32, in series with an additional divide by 2 to 8 in the feedback path. The forward divider is also in series with the divide by 2 to 8. To generate mclk from the PLL output frequency, it has selectable values of 1, 2, and 4, plus a divide by 64 for use during Nap and Deep Nap modes. The forward divider can then generate the data rate clock from mclk with selectable values of 2, 3, 4, 6, and 12 (values of 8 and 24 are also available for debug, but are not supported on the processor interface bus, nor by the frequency scaling facility). Despite these many possible configurations, one constraint that limits the combinations of frequencies that can be used in the 970MP microprocessor is imposed by the *psync* counter.

The *psync* counter in the 970MP microprocessor continuously counts 24 mclks and then resets to zero, except when Dclk values of 4 and 12 are used. In these cases, the counter counts to 48. This *psync* counter is used to generate processor interconnect control signals that are synchronized with the North Bridge drivers and receivers, as mediated by the *psync* signal. Whenever a *psync* pulse is detected, the *psync* counter value is checked to be sure that synchronization is maintained. Because the *psync* pulse occurs once every 24 SYSCLK cycles, the mclk frequency is constrained to be a multiple of the SYSCLK frequency (an even multiple in the case of a Dclk divider of 4 or 12). The frequency scaling capability on the 970MP microprocessor further constrains the clock configuration values, because this *psync* counter constraint applies to the reduced-frequency, as well as the high-frequency, clock rates.

To meet the *psync* counter constraint at high, medium, and low frequencies, the only allowable divider values in the feedback path are multiples of four. With a feedback value of eight, for example, using a forward divider value of one yields the high-frequency mclk that is eight times the SYSCLK. Using a divider value of two then yields the medium-frequency mclk that is four times the SYSCLK. Using a divider value of four yields the low-frequency mclk that is two times the SYSCLK.

There are several constraints on frequency configurations for the 970MP microprocessor besides that imposed by the *psync* counter (see the *Power Management for the PowerPC 970FX RISC Microprocessor Application Note* for details).

9.6 Time-Base and Decrementer Registers

The Time-Base and Decrementer Registers run at a constant frequency, independent of changes to the processor and bus frequencies. The default operation of these timers is to run at 1/16 of the full processor frequency, even when the processor itself is running at a lower frequency. When *tben* is configured to clock these timers (HID0[19] equals '1'), the timers will run at the *tben* frequency. When the external clock input mode is used, the *tben* input frequency must not exceed the value specified in the *IBM PowerPC 970MP RISC Microprocessor Datasheet*.

Because the mesh clock frequency can be lowered to 1/64 of the full-speed, the time base and decrementer can be increased or decreased by more than one at a time. Therefore, testing that the decrementer has reached the value of zero in order to generate an internal interrupt is not sufficient. The logic detects that the counter has wrapped around. Additionally, the time resolution of the counters cannot exceed the mesh clock frequency.

9.7 I^2C Bus Interface

The I^2C bus interface operates at a constant speed independent of the current processor frequency.

Note: No I^2C operations are supported during Deep Nap.

9.8 Frequency and Voltage Scaling

9.8.1 Frequency Scaling

Whenever an application requires less than the maximum performance available from the processor, reducing the processor clock frequency can reduce active power linearly. Furthermore, a reduction in frequency allows a reduction in voltage, resulting in an additional quadratic reduction in active power, plus a reduction in leakage.

Frequency scaling on the 970MP microprocessor involves changing the bus frequency along with the processor frequency, due to the high speed of the processor interconnect bus, and the constraint that the processor frequency be at least twice the bit rate of the bus. In order to support frequency scaling in a multi-processor system, the North Bridge must be involved in initiating the sequence (see the *Power Management for the PowerPC 970FX RISC Microprocessor Application Note* for details).

9.8.1.1 Initiating a Frequency Change

Software initiates a frequency change by writing to the PCR. The value written to the PCR frequency field determines the target frequency being switched to. The values in the parameter fields must correspond to this new frequency. Similarly, if the voltage field is used, the voltage requested must correspond to the frequency requested. The North Bridge is responsible for changing the voltage *before* the frequency change when raising voltage. It is also responsible for changing the voltage *after* the frequency change when lowering the voltage.

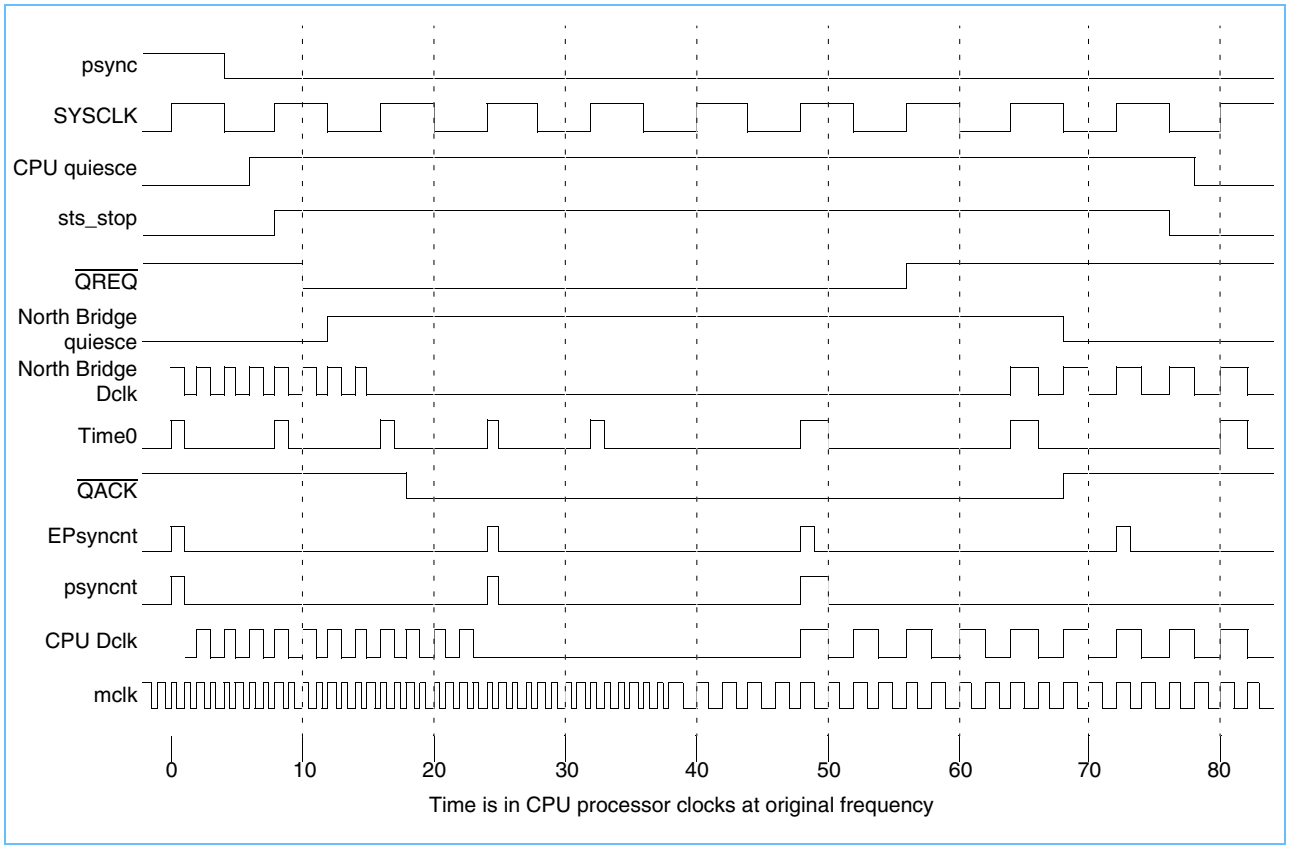
The QREQ and QACK signals have been overloaded to provide handshaking during the frequency change procedure. Therefore, these signals are not available for their normal use (handshaking for Nap mode) during the procedure. System hardware or software must enforce the negation of these signals at the beginning of the procedure. If a processor puts itself into Nap mode during the frequency change procedure, the processor blocks assertion of the QREQ signal for Nap signalling until after the frequency change is complete.

The waveforms in *Figure 9-6* on page 283 show the ordering of events on the CPU-to-North Bridge interface during a frequency change in which the clocks are slowed to half speed. The time shown at the bottom of the figure is in CPU processor clocks at the original frequency. However, this figure is intended to show the ordering of events, and not actual latencies between events. Latencies are discussed in *Section 9.8.5 Frequency and Voltage Scaling Latencies* on page 292.

The sequence in *Figure 9-6* on page 283 starts at the point after a CPU has sent the change request to the North Bridge, and the North Bridge has reflected that request to all the processors. Each CPU then completes any bus transactions in progress, and reach a quiescent state. The CPU quiesce signal shown in *Figure 9-6* indicates that the quiescent state is reached at cycle 6. Two cycles later, the CPU asserts its internal *sts_stop* signal. At this point, the core no longer has access to the L2 cache or bus. Two cycles later, the CPU asserts QREQ. During this time, the North Bridge has also been progressing toward a quiescent state. The North Bridge quiesce signal indicates that this state is reached at cycle 12, though it might occur before QREQ is asserted.

The combination of QREQ asserted and North Bridge quiescent causes the North Bridge to stop its bus clocks on a Time0 boundary, which occurs at cycle 16. Next, the North Bridge asserts QACK, shown at cycle 18. Once QACK is asserted, the CPU stops its bus clocks on the next internal *psync* boundary (psyncnt), shown at cycle 24. With its bus clocks stopped, the CPU changes the frequency of its processor (and therefore bus) clock, shown at cycle 38. Once the frequency change occurs, the CPU will start its bus clocks on the next *psync* boundary, shown at cycle 48. After starting its bus clocks, the CPU will negate QREQ, shown at cycle 56. The North Bridge then starts its bus clocks on a Time0 boundary (cycle 64), after which it negates QACK (cycle 68). Internal to the CPU, the negation of QACK leads to the negation of *sts_stop* (cycle 76). This enables allowing core access to the L2 cache and activity to proceed on the bus.

Figure 9-6. Frequency Scaling Event Ordering



9.8.1.2 Power Control Register

Software writes the PCR bits to indicate that a frequency change is wanted, and to pass the information corresponding to that frequency change to all the processors in the system. Writing to the PCR initiates the frequency change process, by generating a special bus transaction that is sent to the North Bridge and eventually reflected to all the processors. The address bits of this special transaction are copied from PCRH[22:31] and PCR[0:31] (see *Section 9.8.1.3 Power Control Register High (PCRH)* on page 286).

Note: The special bus transaction is generated when the PCR Register is written, so the Power Control Register High (PCRH) must be updated as needed before writing the PCR.

The PCR is implemented as a scan communications (SCOM) register; the odd-parity address for JTAG access is x'0AA0 0100'. The PCRH is also implemented as an SCOM register, at the same address as the PCR. The high-order bit in the register indicates which register is being written. The PCR high-order bit equals '1'; the PCRH high-order bit equals '0'. The 32-bit PCR and PCRH are written using Move To Special Purpose Register (**mtspr**) instructions that target the SCOM data (SCOMD) and SCOM control (SCOMC) special purpose registers (SPRs). Thus, the low-order 32 bits (bits 32:63) of the source register are moved to the target PCR or PCRH.

For example, before initiating a frequency change, set the following registers:

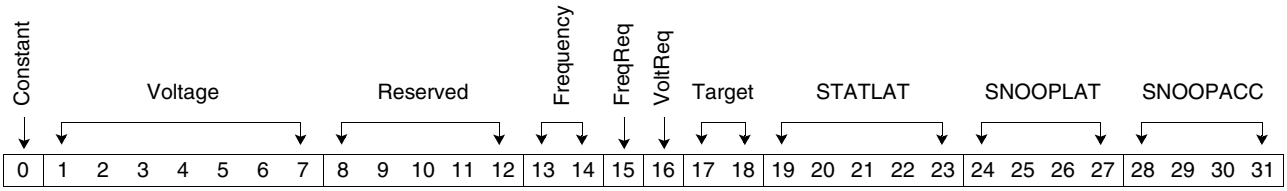
- Appropriate values in the low-order bits of gpr3 to indicate the required settings for the PCR (including bit 32 equals '1')
- Appropriate values for PCRH in gpr4 (including bit 32 equals '0')
- The SCOM address of these registers in gpr5 (to x'0000 0000 0AA0 0100')

The following sequence, where "gpr" stands for General Purpose Register, initiates a frequency change:

```
.set SCOMD 277                          # SPRN for SCOMD
.set SCOMC 276                          # SPRN for SCOMC
mtspr SCOMD, gpr4
isync
mtspr SCOMC, gpr5
isync
mtspr SCOMD, gpr3
isync
mtspr SCOMC, gpr5
```

Note: For the 970MP microprocessor, each frequency change should be preceded by a write to the PCR in which Gd contains all zeros. Not clearing the PCR will prevent further frequency scale commands from being issued by the bus even though the instruction sequence will complete within the processor.

Address x'0AA001'



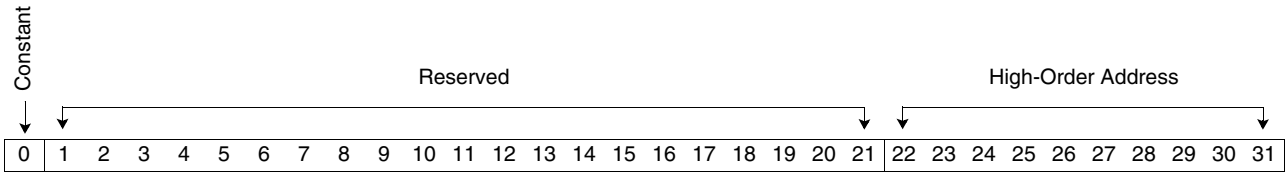
GPR Bits	PCR Bits	Field Name	Description
0	32	Constant	Must be '1'.
1:7	33:39	Voltage	Voltage field.
8:12	40:44	Reserved	Spare field.
13:14	45:46	Frequency	Frequency field. 00 Full frequency 01 Half frequency 10 Quarter frequency 11 Illegal
15	47	FreqReq	Frequency request valid.
16	48	VoltReq	Voltage request valid.
17:18	49:50	Target	Target time.
19:23	51:55	STATLAT[1]	STATLAT is the number of bus beats between the last beat of the address/data (AD) packet and the first beat of the transfer-handshake (TH) packet.
24:27	56:59	SNOOPLAT[1]	SNOOPLAT is the number of bus beats between the last beat of a reflected command packet to the first beat of the individual snoop responses from each of the processors received at the North Bridge.
28:31	60:63	SNOOPACC[1]	SNOOPACC is the number of bus beats between the last beat of the individual snoop response sent from a processor to the first beat of the accumulated snoop response received from the North Bridge. **Note:** SNOOPACC is a 4-bit field. When coded with a value of 1 - 15, the actual value is x + 8. For example, a one in the SNOOPACC field is actually a nine. When a zero is coded in this field, the actual value is 24.

1. See *Table 11-1* on page 371 for information about programmable delay parameters.

9.8.1.3 Power Control Register High (PCRH)

The Power Control Register High (PCRH) contains the high-order address field.

Address x'0AA001'



Bits	Field Name	Description
0	Constant	Must be '0'.
1:21	Reserved	Reserved.
22:31	High-Order Address	High-order address field.

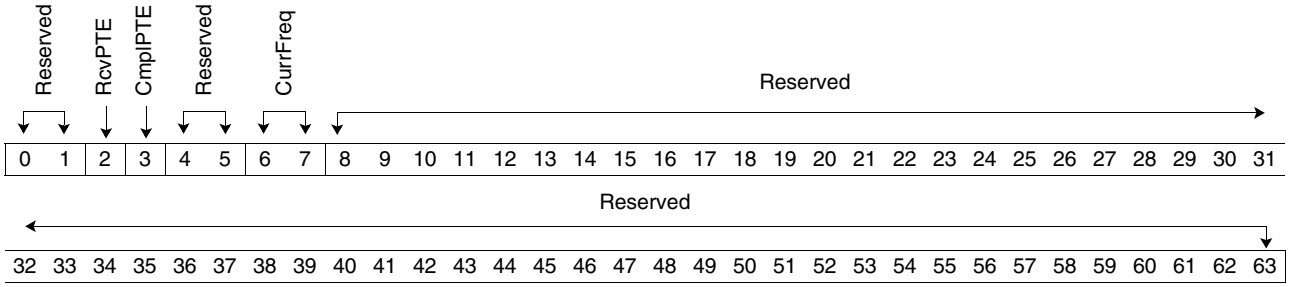
9.8.1.4 Power Status Register

The status of the power tuning facility is available in the Power Status Register (PSR). This register consists of read-only bits, indicating the current voltage (if supported by software) and the current frequency. This frequency value is valid when there is no frequency change in progress, as indicated when PSR[2] equals '0'.

When the processor receives the power adjustment special transaction reflected from the North Bridge, it sets PSR[2] to indicate that a frequency change is in progress. Shortly after the North Bridge has asserted QACK to start the frequency scale, the new frequency field is reflected in PSR[6:7]. Once the frequency scaling has completed PSR[3] is also set to '1'.

An SCOM read of the PSR once bit 2 and 3 are set will automatically clear both bits.

Address x'408001'



Bit	Field Name	Description
0:1	Reserved	Reserved.
2	RcvPTE	Power tuning command has been received.
3	CmplPTE	Power tuning command has completed.
4:5	Reserved	Reserved.
6:7	CurrFreq	Current frequency.
8:63	Reserved	Reserved.

9.8.2 Power Adjustment Bus Transaction

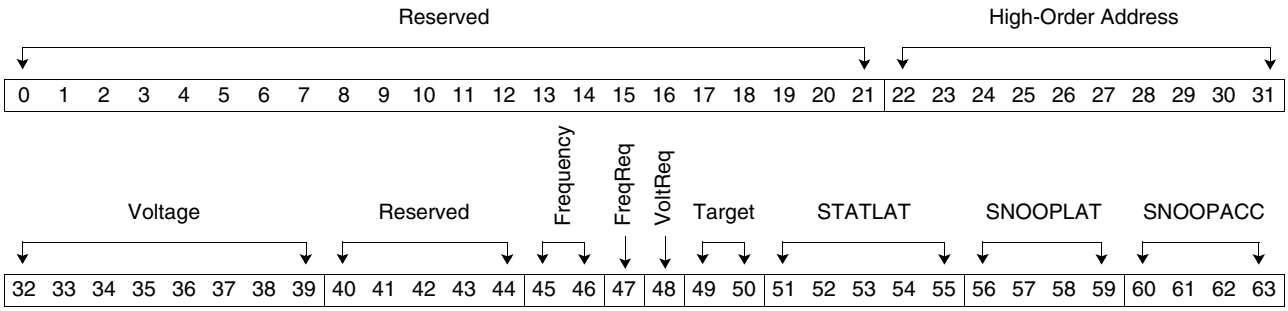
The processor sends a power adjustment transaction to the North Bridge to initiate the frequency and voltage scaling sequence in the system. This is a command-only transaction. It contains information that is encoded in a subset of the address bits to indicate the required target frequency, and the corresponding parameter information. *Table 9-9* shows the transaction type and related bus signals for this transaction.

Table 9-9. Power Adjustment Transaction

Bus Operation	Power Adjustment
Transaction type	0 0101 (x'05')
Address modifiers (WIMGRP)[1]	00 1000
Tag field	1 1011

1. W = write through, I = cache inhibited, M = memory coherent, G = guarded read, R = rerunning, P = pipelined snoop

The encoding of the address bits for this transaction is as follows:



Bits	Field Name	Description
0:21	Reserved	Not implemented.
22:31	High-Order Address	High-order address bits.
32:39	Voltage	Voltage field. The 970MP microprocessor does not use this field.
40:44	Reserved	Spare field.
45:46	Frequency	Frequency field. 00 Full frequency 01 Half frequency 10 Quarter frequency 11 Illegal
47	FreqReq	Frequency request valid.
48	VoltReq	Voltage request valid.
49:50	Target	Target time.

1. See *Table 11-1* on page 371 for information about programmable delay parameters.

Bits	Field Name	Description
51:55	STATLAT[1]	STATLAT is the number of bus beats between the last beat of the AD packet and the first beat of the TH packet.
56:59	SNOOPLAT[1]	SNOOPLAT is the number of bus beats between the last beat of a reflected command packet to the first beat of the individual snoop responses from each of the processors received at the North Bridge.
60:63	SNOOPACC[1]	SNOOPACC is the number of bus beats between the last beat of the individual snoop response sent from a processor to the first beat of the accumulated snoop response received from the North Bridge. **Note:** SNOOPACC is a 4-bit field. When coded with a value of 1 - 15, the actual value is x + 8. For example, a one in the SNOOPACC field is actually a nine. When a zero is coded in this field, the actual value is 24.

1. See *Table 11-1* on page 371 for information about programmable delay parameters.

These 42 low-order address bits are copied from the corresponding fields in the Power Control Register. Software can set bits 22 to 31 to any desired value to make the address fall within some desired range. Furthermore, if the voltage field is unused (voltage request valid is negated), bits 32 to 39 and the spare bits (40 to 44) can also be set to any desired value by software.

The North Bridge uses the frequency field to determine what new frequency is being requested. The frequency-request valid bit must be asserted if a frequency change is being requested. The presence of this bit allows the option of a voltage-change-only request. If the frequency request valid bit is negated, the North Bridge will not reflect this transaction to the processors. It is illegal to issue a frequency scale request to '11' or to the same frequency scale factor (that is, to issue a frequency scale command to full when it is already full). The North Bridge will not reflect these transactions to the processors.

Once the transaction is reflected to the processors, each processor responds as follows:

- If the frequency field indicates no change, the processor does nothing.
- If the frequency field indicates a change to the current frequency, or a change to a new frequency, then the processor must execute the frequency change procedure.

In addition to the four parameters passed in the power adjustment transaction, the processor interconnect also depends on the values of the programmable bit line and clock delays that are determined during the IAP at power-on. To support frequency scaling, this IAP must be run at the high-frequency, high-voltage setting for the processor. Then, the effect of running at lower frequencies is to widen the signal eye. However, the effect of lowering the core voltage while the I/O voltage remains constant is to increase all the bit and clock delays. Thus, once the IAP establishes the minimal bit skew and clock centering required for the interface to run at high frequency and high voltage, these same settings should also support the lower frequencies and voltage. Therefore, there is no facility for changing these delay values during a frequency or voltage change.

9.8.3 Clock Dithering

Input current to the processor can change significantly during transitions of the power tuning frequency. These current changes must be controlled to avoid over and under voltages that a high di/dt might cause due to inductance in the power distribution network. A clock-dithering mechanism included in the power tuning facility enables gradually transitioning between frequencies.

The power tuning facility supports frequency scaling with a constant-frequency PLL that feeds multiple frequency dividers. The outputs of these dividers are fed to a frequency multiplexer, from which one divider output is selected as the processor mesh clock at any given time. Toggling this multiplexer-selection signal during a transition from frequency A to frequency B accomplishes clock dithering. Thus, most clocks are at frequency A at the beginning of the transition. Gradually, more and more frequency-B clocks are introduced in the dithering pattern.

Figure 9-7. Clock Dithering Block Diagram

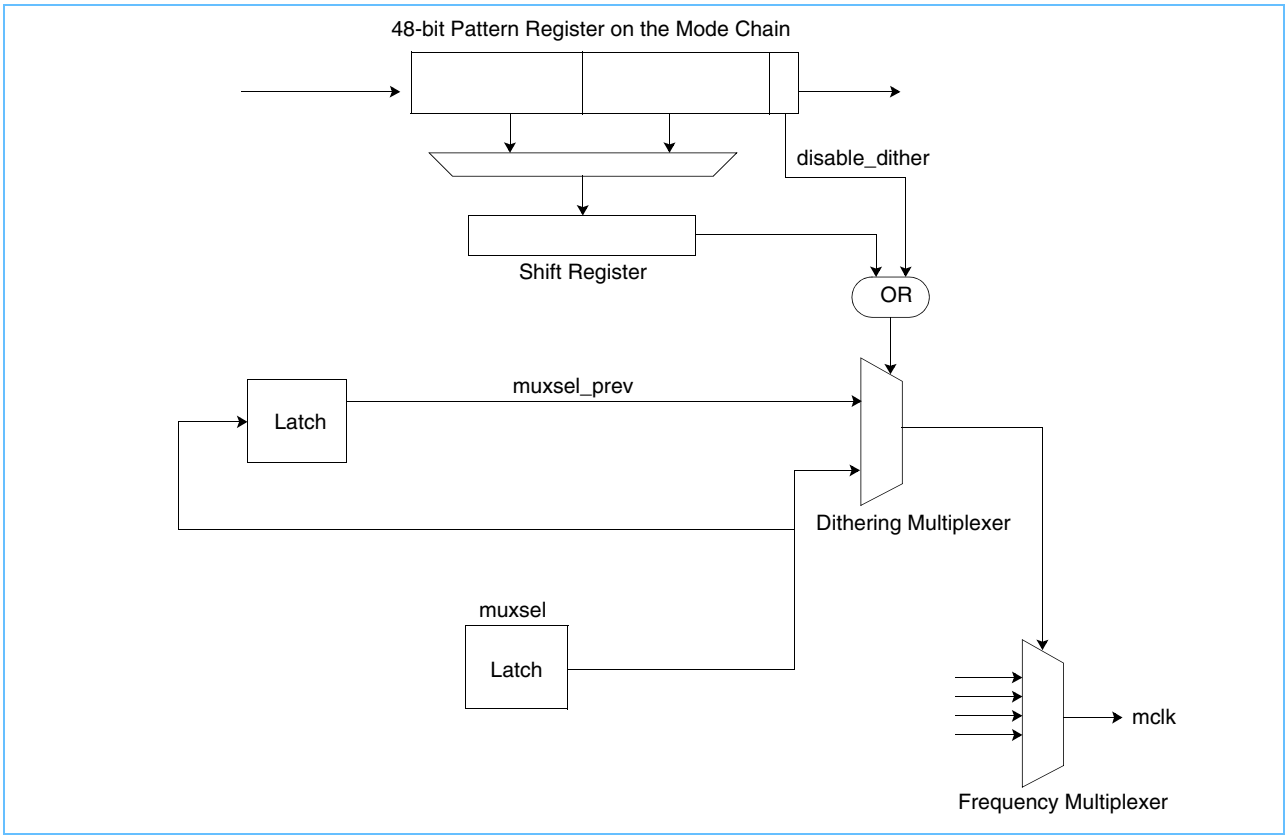


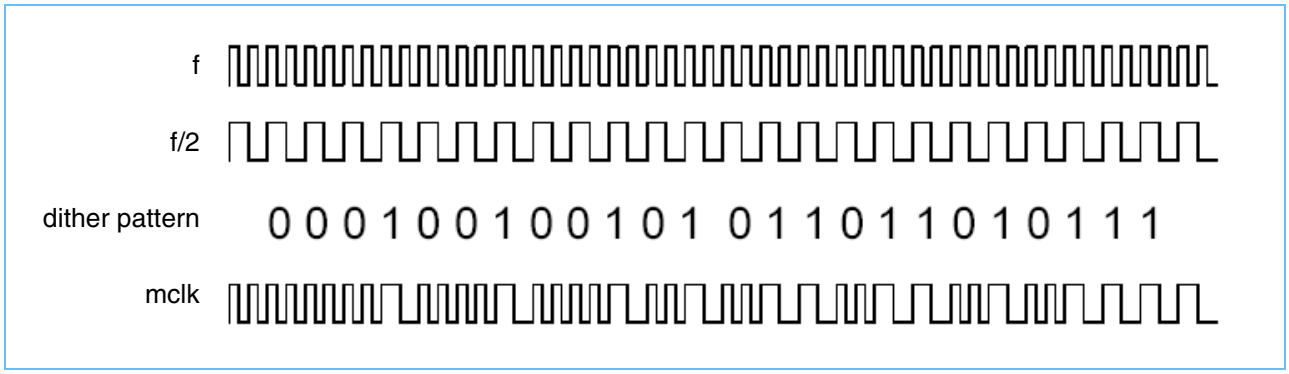
Figure 9-7 shows the components controlling the dithering of the clock. Two dithering patterns, selectable as either 24 or 48 bits in length, are provided in the mode ring. They support distinct dithering patterns for transitions between the high and medium frequencies and the transitions between the medium and low frequencies. When a frequency shift is initiated, the appropriate mode ring pattern is selected using a multiplexer for transfer to a 24-bit shift register. At the same time, the multiplexer select pattern for the previous frequency is saved in the *muxsel_prev* latch, while the new frequency is loaded into the *muxsel* latch.

Clock dithering involves a 2-level multiplexer selection process. The Shift Register is clocked at the lower of the previous and new frequencies. Starting on the rising edge of the mclk/4, it shifts the pattern one bit to the right every cycle. Then it applies the right-most bit to the dithering multiplexer to select a multiplexer-selection

pattern. That pattern is then applied to the frequency multiplexer to select the mesh clock frequency. A '1' bit in the shift pattern selects the new frequency; a '0' bit selects the old frequency. At the end of the shift pattern, a '1' bit is forced to continuously select the new frequency. A separate mode-ring bit can be used to disable clock dithering by forcing this control bit to always be a '1' through the OR circuit shown in *Figure 9-7*.

As an example of a shift pattern for achieving a gradual transition from high to medium frequency might be '1110 1011 0110 1010 0100 1000' (see *Figure 9-8* on page 291). These bits are shifted at the medium frequency. Each '1' corresponds to one cycle of medium frequency. Each '0' corresponds to two cycles of high frequency (because the Shift Register is clocked at medium frequency). Thus, reading the pattern from right to left, the pattern specifies six fast clocks, followed by one medium clock, followed by four fast clocks, followed by one medium clock, and so on, as indicated in the *Figure 9-8*.

Figure 9-8. Sample Shift Pattern



9.8.4 Voltage Scaling

To take the greatest advantage of frequency scaling, it is desirable to vary the voltage to match processing requirements. In operational modes, when the frequency is reduced, the voltage can also be reduced to realize a quadratic reduction in active power. When the voltage is changed with frequency, the voltage change must precede frequency increases, and must follow frequency decreases.

The processor supports the integration of voltage and frequency scaling as currently described. However, the software and system designers might choose to control the processor voltage independently of the power tuning facility. In that case, the software and system will be responsible for the sequencing and timing of voltage changes with respect to frequency changes (see the *Power Management for the PowerPC 970FX RISC Microprocessor Application Note* for details).

9.8.5 Frequency and Voltage Scaling Latencies

The sequence for raising the voltage and frequency has a latency from the time the operating system writes the configuration value in the PCR to the time when the status bit in the PSR indicates that the change is complete. That latency has the following components:

- Time to signal North Bridge
- Time to raise voltage
- Time to signal processors
- Time for North Bridge and processors to quiesce
- Time for North Bridge and processors to handshake
- Time for one *psync* (1:24) cycle
- Time to handshake and reset the status bit

The latency for lowering the frequency and voltage is similar. However, the voltage is lowered after the frequency, and processing does not need to wait for that to occur.

While the processor signals the frequency change to the North Bridge, and until the North Bridge reflects the power adjustment command to the processor, it proceeds normally. Once the processor begins to quiesce the bus, the processor core will no longer be able to access data and instructions from the L2 or bus interface. As long as the processor is able to execute with data and instructions in the L1 caches, it can continue to run. In the best case, the processor will only stall for about a cycle when the mesh clock frequency itself is switched. More specifically, the processor will be unable to respond to interrupts while the bus interface is in a quiescent state, unless the instructions and data needed to handle the interrupt are in the L1 caches before the frequency change. This means the interrupt response might be delayed due to a frequency switch (see the *IBM PowerPC 970MP RISC Microprocessor Datasheet* for latency values).

9.9 Reducing Clock Mesh Power

There are two power saving modes defined for the 970MP microprocessor, Nap and Doze. In Nap mode, the clocks to the core are turned off, while the timers, PLL, and part of the pervasive unit continue to operate. Doze mode is similar, except that snoop logic is also active. Doze mode is entered from Full Power (Full Run) mode by setting HID0[nap], and then the MSR[POW] bit. This also causes the QREQ signal to be asserted, requesting that the North Bridge put the bus in a quiescent state. When the North Bridge complies, it asserts QACK, causing the processor to transition into Nap mode. Whenever QACK is negated, the processor must return to Doze mode to process snoop transactions.

9.9.1 Power Saving in Deep Nap

When the processor is in Nap mode, the core is inactive, and clocks are gated at local clock buffers (LCBs). However, the clock distribution mesh itself continues to be clocked, dissipating significant power. To reduce the active power during Deep Nap mode, the processor clock is divided down to a very low frequency. The frequency is then be brought back up to its functional level as the processor transitions out of Deep Nap mode.

The frequency switching for Nap mode is completely under hardware control. When enabled, the frequency switch takes place after the clocks have been gated and subsequent to detecting that QACK has been asserted. Entering low-frequency, Deep Nap mode takes only one cycle longer than entering Nap mode without changing frequency. During low-frequency Nap mode, the bus clocks are disabled, and the bus signals are driven with the null transaction pattern.

As with normal Nap mode, negation of QACK or detection of an external (or decrementer, *hreset*, *sreset*, or machine check) interrupt causes the processor to leave low-frequency Nap mode. The time required to exit low-frequency Nap mode is longer than the time to exit normal Nap mode, due to the frequency change and corresponding synchronization required. Latencies for the transitions from Deep Nap to Doze and Full Run modes can be found in the *IBM PowerPC 970MP RISC Microprocessor Datasheet*.

HID0[deep nap] controls whether the clock frequency is reduced during Nap mode. When the bit is asserted, the processor will transition to Deep Nap mode immediately after entering Nap mode.

9.10 Additional Dynamic Power Management

The 970MP microprocessor implements dynamic power management—the gating of clocks to idle circuits while in an operational mode—in a number of functional units. For example, there are two levels of clock control for the VPU, a coarse level and a fine level. The coarse control is essentially a static form of clock gating control, making use of the vector processor available bit (MSR[VP]). When this bit is a zero, the latches in all VPU stages from issue to writeback are gated off. The fine level control is much more dynamic. It occurs on a stage-by-stage basis within each execution pipeline, starting with the latches following stage 2 of the Vector Register File (RF2). When this fine level of control is enabled, all clocks in all of the VPU stages from the register access to the write back are gated off at all times. The only exceptions are cycles during a stage that has active instructions.

Clock gating has been implemented in the VPU, IDU, STS, ISU, FXU, FPU and pervasive units. Because DPM has no negative impact on performance, it should always be enabled. For test purposes, DPM can be disabled as follows:

- For the VPU, IDU, and STS units, DPM is disabled by negating HID0[DPM].
- For the ISU, FXU, FPU, and pervasive units, setting bit 0 in the Dynamic Power-Management Options Register (x'000800') to a '1' disables DPM.

10. 970MP Performance Monitor

The 970MP microprocessor has a complex, speculative, out-of-order execution core coupled with an equally complex multilevel storage hierarchy. Users concerned with performance analysis and system optimization have access to performance monitoring features, which support a wide range of tasks which include:

- Profiling memory hierarchy behavior and tuning system algorithms to optimize scheduling, partitioning, and structuring for tasks and data
- Tuning applications for the target system
- Debugging, analyzing, and optimizing processor architecture features

The performance monitor facility provides information for a wide variety of activities and is part of the facilities that are collectively referred to as instrumentation facilities. Instrumentation facilities include matching/sampling, tracing, and thresholding.

Note: The 970MP performance monitor should only be used as a debug facility until characterization of its features and functions is complete.

10.1 Instrumentation Facilities Overview

The 970MP performance monitoring facility is an extension to that of earlier PowerPC processors. There are eight Performance Monitor Counter Registers (PMC1-8). They can count a variety of events, many of which are relevant to performance analysis. As before, the counters support user or supervisor and marked or unmarked filtering of events. A marked instruction is one that is eligible for sampling as determined by the instruction fetch unit (IFU) and instruction dispatch unit (IDU) instruction matching facilities.

The most-significant change introduced by the 970MP performance monitor is the concept of indirect events. A subset of the normally selected direct Performance Monitor Counter (PMC) events are multiplexed so that there is a larger number of total available events. Unlike event selection on previous PowerPC processors (which had only direct events), indirect events cannot be configured entirely independently (setting a multiplexer affects the indirect events on more than one PMC). Some indirect events can also be summed together by the hardware. This feature is most often used to sum the performance event counts of a functional unit pair (for example, floating-point unit 0 [FPU0] and floating-point unit 1 [FPU1]).

10.1.1 Performance Monitor Facilities

The instrumentation performance monitor (perfmon) on the 970MP microprocessor includes the following functions:

- Counts up to eight concurrent software selected events in individual 32-bit counters. The counting of events can be enabled by software under several conditions such as user (problem) or supervisor (privileged) state, and Run or Wait state.
- Generates a maskable exception when an event counter overflows (triggering).
- Freezes the contents of the event counters until a selected trigger occurs and then begin counting (triggering).
- Increments the event counters until a selected trigger occurs and then freezes counting (triggering).
- Monitors classes of instructions selected by the instruction matching facility.
- Randomly chooses an instruction for detailed monitoring (sampling).
- Counts start/stop event pairs that exceed a selected timeout value (thresholding).

10.1.2 Performance Monitor Event Selection

One event per counter can be selected for monitoring at a given time. The event to be monitored is selected by setting the appropriate value in the Monitor Mode Control Register (MMCR) bit field for that counter. The events counted might be the number of cycles that the event occurs or the number of occurrences of the event depending on the particular event selected.

10.1.3 Machine States and Enabling the Performance Monitor Counters

Performance monitor counting can be enabled or disabled under several machine states, which are selected using the counting control bit fields in the MMCRs and the state bits in other Special Purpose Registers (SPRs).

10.1.4 Trigger Events and Enabling the Performance Monitor Counters

Certain kinds of conditions and events, called trigger events, can be used to control performance monitor activities such as starting or stopping the counters and causing performance monitor exceptions. These scenarios are selected using the condition/event enable bits fields and the exception enable bits of the MMCRs in conjunction with control bits in other SPRs.

10.1.5 Performance Monitor Exceptions

Trigger events can cause performance monitor exceptions to occur based on the values of the exception enable bits in the MMCRs. An enabled exception might cause a performance monitor exception to occur if the exception is enabled in other SPRs.

10.1.6 Sampling

The 970MP microprocessor can be configured to sample instructions for detailed monitoring. The 970MP microprocessor instrumentation facilities support setting mask values for matching particular instructions or kinds of instructions that are then eligible to be sampled (that is, they are marked for sampling). The performance monitor includes events for counting marked instructions at each stage of the pipeline and in certain other situations. Instruction sampling is a useful facility for gathering both detailed and statistical information for particular instructions.

Note: Instruction marking is entirely separate from thread marking with the performance monitor mode bit in the Machine State Register (MSR[PMM]).The state of the MSR[PMM] bit is only relevant for event counting in order to determine when counters should be frozen (MMCR0[FCM1, FCM0] fields).

10.1.7 Thresholding

Unlike previous PowerPC processors, which implemented thresholding only on load instructions, the 970MP processing unit monitors the pipeline stage progression of sampled instructions and can detect when the stage-to-stage cycle count for a selected start/stop pair of pipeline stages exceeds a specified threshold value.

10.1.8 Trace Support Facilities

The 970MP microprocessor supports both the single step and the branch trace modes as defined by the PowerPC Architecture.

10.2 Instruction Sampling Facilities

10.2.1 Special Purpose Registers and Fields Associated with Instrumentation

The 970MP microprocessor instrumentation facilities and associated 970MP microprocessor components include several SPRs used for or associated with performance monitoring, matching, sampling, and tracing. Unless otherwise noted, the Special Purpose Registers described below and listed in *Table 10-1* on page 299 can be read in user (problem) and supervisor (privileged) state by using the Move From Special Purpose Register (**mfspr**) and written in supervisor state by using the Move To Special Purpose Register (**mtspr**) instructions. The MSR Register is read and written by the Move From Machine State Register (**mfmsr**) and Move To Machine State Register (**mtmsr**) instructions.

The 970MP microprocessor instrumentation facilities include the following Special Purpose Registers and register bit fields (also listed in *Table 10-1* on page 299):

- Performance Monitor Mode Control Registers (MMCRx)
  These registers include both counting control and event select bit fields.
- Performance Monitor Counter Registers (PMCx)
  These registers increment each time (or cycle, depending on the selected event) that an event occurs while the counter is enabled. These registers also have the control function for the counter overflow condition.
- Machine State Register [EE] (MSR[EE])
  This register bit is used to enable or disable external interrupts. The performance monitor exception is considered an external interrupt.

- Machine State Register [PMM] (MSR[PMM])
  This register bit is used to enable or disable performance monitor activity controlled by the process mark bit.
- Machine State Register [PR] (MSR[PR])
  This register bit is used to establish user (problem) or privileged (supervisor) mode and the performance monitor counting activity controlled by this bit.
- Machine State Register [SE] (MSR[SE])
  This register bit is used to enable or disable the trace exception after each instruction is completed.
- Machine State Register [BE] (MSR[BE])
  This register bit is used to enable or disable the branch trace exception and after a branch instruction is completed.
- Hardware Implementation-Dependent Register0[13] (HID0[TG])
  This register bit is used to determine the granularity the thresholder uses for counting cycles.
- Control Register[31] (CNTL[31])
  This register bit is used to determine the Wait or Run state and the performance monitor activity controlled by this bit.
- Scan Communication Register x'240' [0:15] (SCOM x'240' [0:15])
  These register bits are used to establish the timeout and resume delays used by the performance monitor to coordinate the matching and sampling facility.
- Scan Communication Register x'340' [11:12] (SCOM x'340' [11:12])
  These register bits are used to establish the matching and sampling filter mode used by the matching and sampling facility to produce marked instructions that can be counted by the performance monitor.
- Instruction Match Content-Addressable Memory (CAM) Registers (IMC)
  The IMC SPRs are used to access the IMC array that contains the mask values used for instruction matching. The Move To IMC (**mtimc**) and Move From IMC (**mfimc**) instructions can be executed only in supervisor mode.
- Time-Base Register [47, 51, 55, 63] (TB[47, 51, 55, 63])
  These register bits are used to enable or disable the time-base events that can be used to enable or disable performance monitor counting.
- Sample Address Registers (SxAR)
  The Sampled Instruction Address Register (SIAR) contains the address and the Sampled Data Address Register (SDAR) contains the data relating to a marked instruction. The registers can be read in supervisor (privileged) or user (problem) state, but are modified only by the hardware. The values written to these registers by the hardware depend on the processing state and on the kind of instruction that is being marked for sampling.
- Machine Status Save/Restore Register (SRR0, SRR1)
  These registers are used to save machine status during exception handling. In addition, SRR1[33] is used to determine when the contents of the SIAR and SDAR Registers are synchronized, so that they refer to the same marked instruction.

Table 10-1. 970MP Performance Monitor and Trace-Related Special Purpose Registers

Register Name	SPR Address Bits[1]		Function
	5:9	0:4[2]	
MMCR0	'11000'	'n1011'	Performance Monitor Mode Control Register 0
MMCR1	'11000'	'n1110'	Performance Monitor Mode Control Register 1
MMCRA	'11000'	'n0010'	Performance Monitor Mode Control Register A
PMC1	'11000'	'n0011'	Performance Monitor Counter Register 1
PMC2	'11000'	'n0100'	Performance Monitor Counter Register 2
PMC3	'11000'	'n0101'	Performance Monitor Counter Register 3
PMC4	'11000'	'n0110'	Performance Monitor Counter Register 4
PMC5	'11000'	'n0111'	Performance Monitor Counter Register 5
PMC6	'11000'	'n1000'	Performance Monitor Counter Register 6
PMC7	'11000'	'n1001'	Performance Monitor Counter Register 7
PMC8	'11000'	'n1010'	Performance Monitor Counter Register 8
MSR[61]	Use **mtmsr**, **mfmsr** instructions (supervisor [privileged] mode only)		Machine State Register [Performance Monitor Mark]
MSR[48]			Machine State Register [External Interrupt]
MSR[49]			Machine State Register [User (Problem)/Supervisor (Privileged) State]
MSR[53]			Machine State Register [Single-Step Trace Enable]
MSR[54]			Machine State Register [Branch Trace Enable]
HID0[13]	'11111'	'10000'	Hardware Implementation-Dependent Register 0 [Threshold Granularity]
CTRL[31]	'00100'	'n1000'	Control Register [Run Bit]
SCOMC	Use **mtscomc**/**d** and **mfscomc**/**d** instructions		Scan Communication Control
SCOMD			Scan Communication Data
IMC	Use **mtimc**, **mfimc** instructions (supervisor mode write, user and supervisor mode read)		Instruction Match CAM Register
TBL [47,51,55,63]	'01000'	'n1100'	Time-base bits used for performance monitor time-base events
SIAR	'11000'	'n1100'	Sampled Instruction Address Register
SDAR	'11000'	'n1101'	Sampled Data Address Register
SRR1	'00000'	'n1011'	Machine Status Save/Restore Register 1

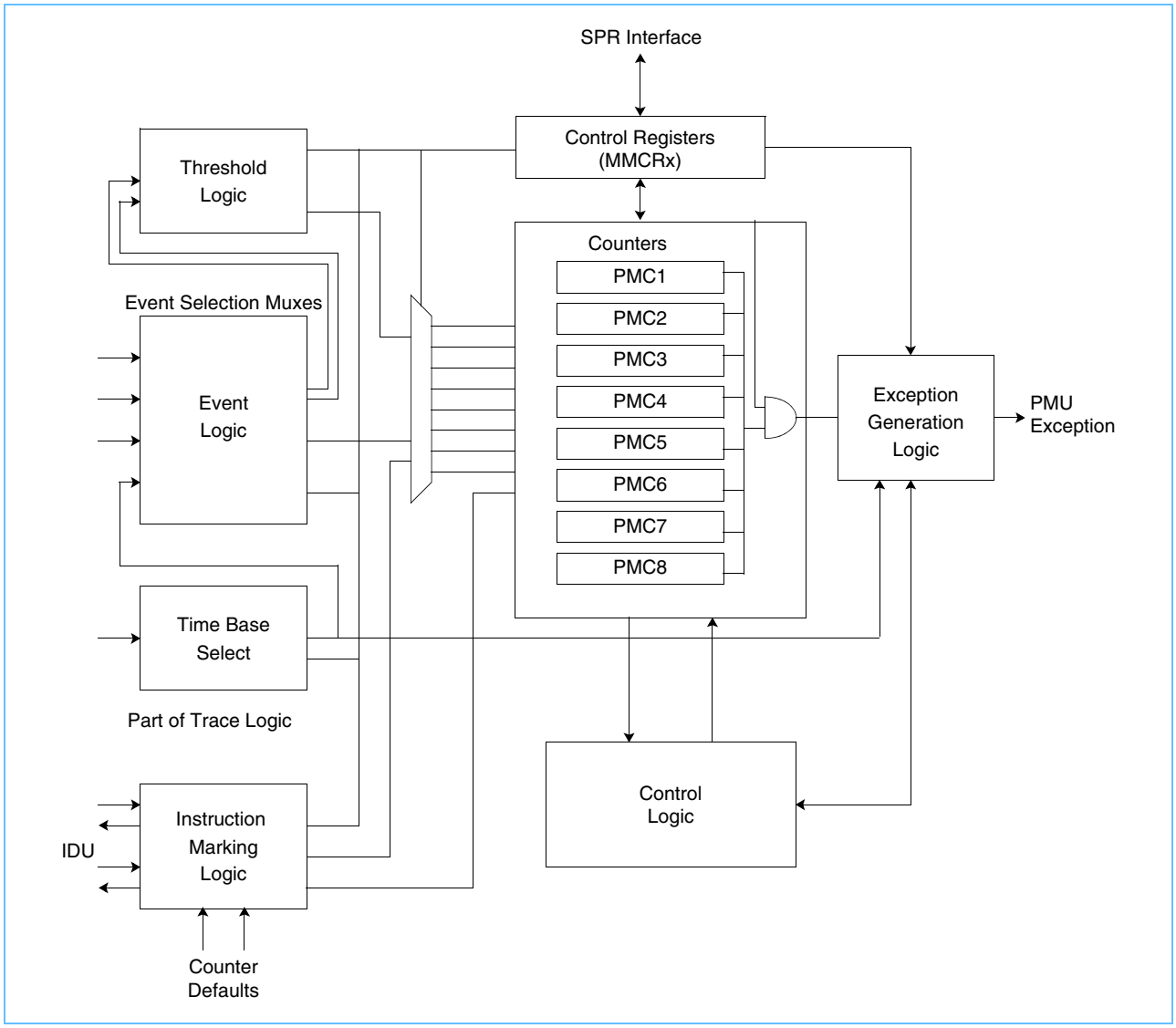
Note:

1. In a **mtspr**/**mfspr** instruction, the instruction SPR field of bits 11:15 hold SPR address bits 0:4 and bits 16:20 hold SPR field bits 5:9.
2. When n is set to '1', it indicates an SPR address value for a supervisor mode **mtspr** or **mfspr** instruction.
   When n is set to '0', it indicates an SPR address value for a user mode **mfspr** instructions.
   For **mfspr**, the instruction is supervisor mode if and only if SPR[0] is set to '1'.

10.3 Performance Monitor Components

A schematic overview of the components that make up the 970MP performance monitor is shown in *Figure 10-1*. These components and their use are described in the following sections.

Figure 10-1. Performance Monitor Architecture



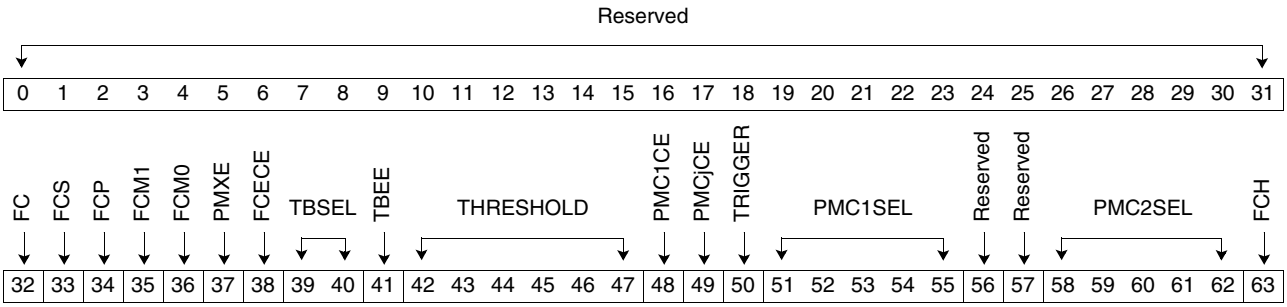
10.4 Performance Monitor Control Registers

The Performance Monitor Control Registers, MMCR0, MMCR1, and MMCRA, are used in conjunction with the MSR and other SPRs to set up the performance monitor enable states, exception conditions, threshold values, match criteria, and selection of the events counted in each of the Counter Registers, PMC1 - PMC8.

The MMCRx Register bit assignments are shown in *Section 10.4.1 Performance Monitor Control Register MMCR0* on page 301, *Section 10.4.2 Performance Monitor Control Register MMCR1* on page 304, and *Section 10.4.3 Performance Monitor Control Register MMCRA* on page 307. The MSR bits that relate to performance monitor functions are shown in *Table 10.4.5 Performance Monitor and Trace Related Bits in the Machine State Register (MSR)* on page 310.

For all of the Performance Monitor Control Register fields, it is always understood that the counter is incremented if that action is not prohibited by some other control condition. All of the MMCRx and PMCx Registers flush to zero unless otherwise noted in the following MMCRx and PMCx tables.

10.4.1 Performance Monitor Control Register MMCR0

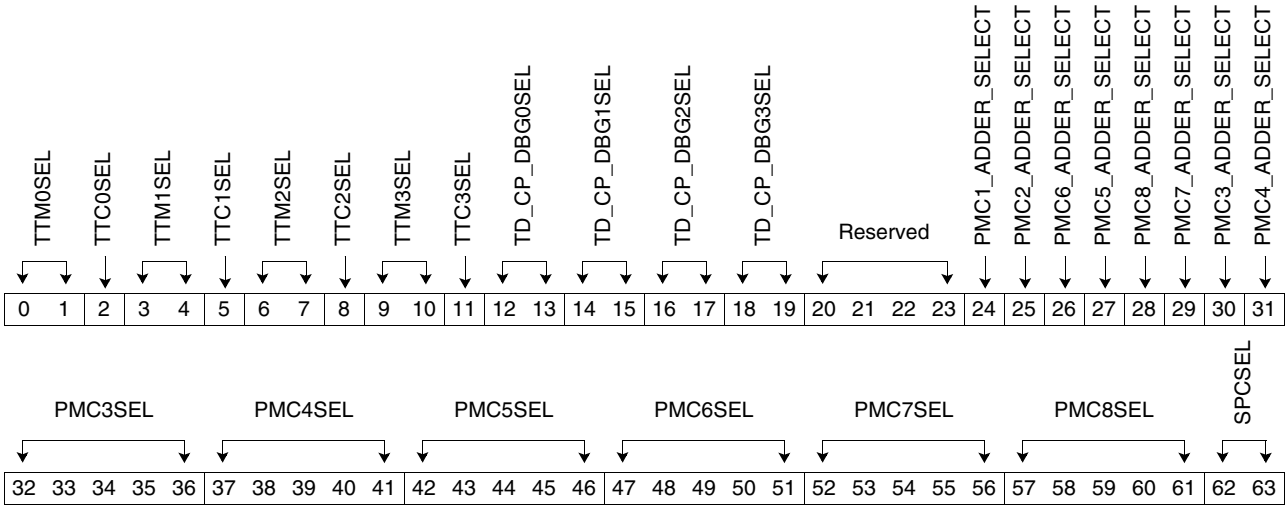


Bits	Field Name	Description
0:31	—	Reserved.
32	FC	Freeze counters. 0 The PMCs are incremented. 1 The PMCs are not incremented. The processor sets this bit to '1' when an enabled condition or event occurs *and* the "freeze counters on enabled condition or event" bit is '1' (MMCR0[FCECE] = '1').
33	FCS	Freeze counters when in supervisor state. 0 The PMCs are incremented. 1 The PMCs are not incremented in supervisor state (MSR[PR] = '0').
34	FCP	Freeze counters when in user (problem) state. 0 The PMCs are incremented. 1 The PMCs are not incremented in user (problem) state (MSR[PR] ='1').
35	FCM1	Freeze counters when performance monitor mark bit (MSR[PMM]) is set to '1'. 0 The PMCs are incremented. 1 The PMCs are not incremented when the MSR mark bit is '1' (MSR[PMM] = '1').
36	FCM0	Freeze counters when performance monitor mark bit (MSR[PMM]) is set to '0'. 0 The PMCs are incremented. 1 The PMCs are not incremented when the MSR mark bit is '0' (MSR[PMM] = '0').

Bits	Field Name	Description
37	PMXE	Performance monitor exception enable. 0 Performance monitor exceptions are disabled. 1 Performance monitor exceptions are enabled until a performance monitor exception occurs, at which time the hardware disables the performance monitor exception (MMRC0[PXME] is set to '0'). For implementations that do not provide a performance monitor exception, software can set PXME to '1' and then poll the bit to determine whether an enabled condition or event has occurred.
38	FCECE	Freeze counters on enabled condition or event. 0 The PMCs are incremented. 1 The PMCs are incremented until detection of an enabled counter negative condition *or* detection of an enabled time-base transition event occurs *and* the trigger bit enables the detected event (MMCR0[TRIGGER] equals '0'). At that time the counters are frozen (MMCR0[FC] is set to '1') until the condition is reset by software. If the enabled condition or event occurs when MMCR0[TRIGGER] equals '1', then the FCECE bit is treated as if it were '0'.
39:40	TBSEL	Time-base selector. 00 Time-base bit 63 is selected. 01 Time-base bit 55 is selected. 10 Time-base bit 51 is selected. 11 Time-base bit 47 is selected. When the selected time base transitions from '0' to '1' *and* the time-base event is enabled (MMCR0[TBEE] equals '1') *and* the performance monitor exception is enabled, a performance monitor exception occurs and the performance monitor exception is disabled (MMRC0[PXME] is set to '0'). In multiprocessor systems with the Time-Base Registers synchronized among the processors, time-base transition events can be used to correlate the performance monitor data obtained by the several processors provided that software has specified the same TBSEL value for all of the processors in the system. The frequency of the time base is implementation dependent, and a system service routine should be invoked to obtain the frequency before a value for TBSEL is chosen.
41	TBEE	Time-base exception enable. 0 Disable time-base transition events. 1 Enable time-base transition events.
42:47	THRESHOLD	Threshold value. When a threshold event is selected, counting occurs only for those of the selected event occurrences whose duration in number of cycles exceeds the value in the THRESHOLD field.
48	PMC1CE	PMC1 count enable. This bit determines whether the counter negative condition due to a negative value in PMC1 is enabled. 0 Disable PMC1 counter negative condition. 1 Enable PMC1 counter negative condition.
49	PMCjCE	PMCj count enable (where j represents any counter from 2 to 8). This bit determines whether the counter negative condition due to a negative value in PMCj ($2 \le j \le 8$) is enabled. 0 Disable PMCj ($2 \le j \le 8$) counter negative condition. 1 Enable PMCj ($2 \le j \le 8$) counter negative condition.
50	TRIGGER	Trigger enable. 0 The PMCs are incremented. 1 PMC1 is incremented. The PMCjs ($2 \le j \le 8$) are not incremented until PMC1 is negative *or* an enabled condition or event occurs. At that time, the PMCj counters ($2 \le j \le 8$) resume counting and the trigger is disabled (MMCR0[TRIGGER] set equal to '0').
51:55	PMC1SEL	PMC1 event selector. The value in this bit field combined with MMCR1[0:31] determines which event will be counted by PMC1.
56	—	Reserved.
57	—	Reserved.

Bits	Field Name	Description
58:62	PMC2SEL	PMC2 event selector. The value in this bit field combined with MMCR1[0:31] determines which event will be counted by PMC2.
63	FCH	Freeze counters in hypervisor mode.

10.4.2 Performance Monitor Control Register MMCR1

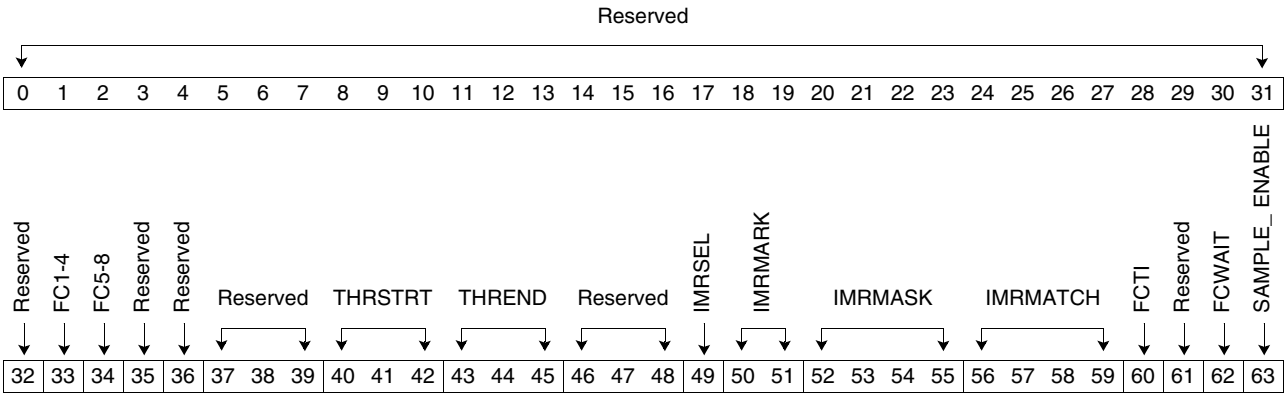


Bits	Field Name	Description
0:1	TTM0SEL	FPU/ISU/IFU/VPU unit select. 00 FPU 01 Instruction sequencer unit (ISU) 10 IFU 11 Vector processing unit (VPU)
2	TTC0SEL	Reserved.
3:4	TTM1SEL	IDU/ISU/STS unit select. 00 IDU 01 Undefined 10 ISU 11 Storage subsystem (STS)
5	TTC1SEL	Reserved.
6:7	TTM2SEL	Reserved.
8	TTC2SEL	Reserved.
9:10	TTM3SEL	Load/store unit 1 (LSU1) select. 0x Lane 2 is LSU1 upper 1x Lane 2 is LSU1 lower x0 Lane 3 is LSU1 upper x1 Lane 3 is LSU1 lower
11	TTC3SEL	Reserved.
12:13	TD_CP_DBG0SEL	Byte lane 0 unit select. 00 Unit from TTM0 01 Unit from TTM1 10 LSU0, byte 0 11 LSU1, byte 0
14:15	TD_CP_DBG1SEL	Byte lane 1 unit select. 00 Unit from TTM0 01 Unit from TTM1 10 LSU0, byte 1 11 LSU1, byte 1

Bits	Field Name	Description
16:17	TD_CP_DBG2SEL	Byte lane 2 unit select. 00 Unit from TTM0 01 Unit from TTM1 10 LSU0, byte 2 11 LSU1, byte 2 or byte 6 (controlled by TTM3SEL[0])
18:19	TD_CP_DBG3SEL	Byte lane 3 unit select. 00 Unit from TTM0 01 Unit from TTM1 10 LSU0, byte 3 11 LSU1, byte 3 or byte 7 (controlled by TTM3SEL[1])
20:23	—	Reserved.
24	PMC1_ADDER _SELECT	PMC1 event adder lane select. 0 Byte lane 0: Add 0 + 4 1 Byte lane 2: Add 0 + 4
25	PMC2_ADDER _SELECT	PMC2 event adder lane select. 0 Byte lane 0: Add 1 + 5 1 Byte lane 2: Add 1 + 5
26	PMC6_ADDER _SELECT	PMC6 event adder lane select. 0 Byte lane 0: Add 2 + 6 1 Byte lane 2: Add 2 + 6
27	PMC5_ADDER _SELECT	PMC5 event adder lane select. 0 Byte lane 0: Add 3 + 7 1 Byte lane 2: Add 3 + 7
28	PMC8_ADDER _SELECT	PMC8 event adder lane select. 0 Byte lane 1: Add 0 + 4 1 Byte lane 3: Add 0 + 4
29	PMC7_ADDER _SELECT	PMC7 event adder lane select. 0 Byte lane 1: Add 1 + 5 1 Byte lane 3: Add 1 + 5
30	PMC3_ADDER _SELECT	PMC3 event adder lane select. 0 Byte lane 1: Add 2 + 6 1 Byte lane 3: Add 2 + 6
31	PMC4_ADDER _SELECT	PMC4 event adder lane select. 0 Byte lane 1: Add 3 + 7 1 Byte lane 3: Add 3 + 7
32:36	PMC3SEL	PMC3 event selector. The value in this bit field combined with MMCR1[0:31] determines which event will be counted by PMC3.
37:41	PMC4SEL	PMC4 event selector. The value in this bit field combined with MMCR1[0:31] determines which event will be counted by PMC4.
42:46	PMC5SEL	PMC5 event selector. The value in this bit field combined with MMCR1[0:31] determines which event will be counted by PMC5.
47:51	PMC6SEL	PMC6 event selector. The value in this bit field combined with MMCR1[0:31] determines which event will be counted by PMC6.
52:56	PMC7SEL	PMC7 event selector. The value in this bit field combined with MMCR1[0:31] determines which event will be counted by PMC7.
57:61	PMC8SEL	PMC8 event selector. The value in this bit field combined with MMCR1[0:31] determines which event will be counted by PMC8.

Bits	Field Name	Description
62:63	SPCSEL	Speculative count event selector. 00 Reserved 01 Event A1x 10 Event A2x 11 Event A3x See *Table 10-6* on page 323 for definitions of the events.

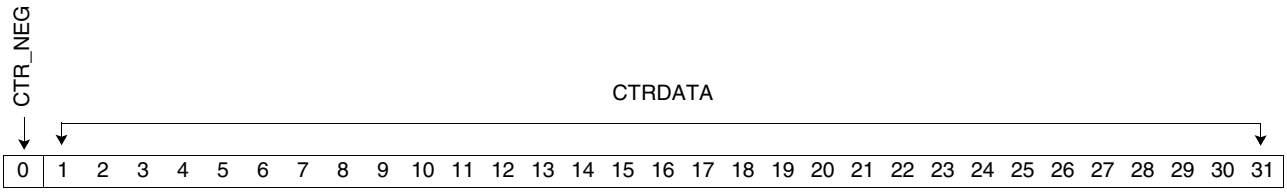
10.4.3 Performance Monitor Control Register MMCRA



Bits	Field Name	Description
0:31	—	Reserved.
32	—	Reserved.
33	FC1-4	Freeze counters 1 - 4. 0 PMC1 - 4 are incremented. 1 PMC1 - 4 are not incremented.
34	FC5-8	Freeze counters 5 - 8. 0 PMC5 - 8 are incremented. 1 PMC5 - 8 are not incremented.
35	—	Reserved.
36	—	Reserved.
37:39	—	Reserved.
40:42	THRSTRT	Threshold start event.
43:45	THREND	Threshold end event.
46:48	—	Reserved.
49	IMRSEL	Instruction mark (IMR) select. IMR select interacts with IMR mark to determine stage 1 eligibility as described in *Section 10.11 IDU Instruction Sampling Facility* on page 344. 0 Stage 1 eligible instructions are determined through predecode bits from the IFU combined with the IMRMATCH and IMRMASK fields as described in *Section 10.11* on page 344. This is useful if the IMR mark equals '00'. 1 The instruction mark bit (IMR bit) from the IFU IMC match array is used to determine Stage 1 eligibility.
50:51	IMRMARK	IMR Mark. Chooses the mark mode for which instructions are Stage 2 eligible. 00 All Stage 1 eligible internal operations (IOPs). 01 Only Stage 1 eligible IOPs that resulted from microcode expansion. 10 Only one IOP per eligible PowerPC instruction. 11 First IOP that goes to the LSU for every eligible PowerPC load/store (**ld**/**st**) instruction.
52:55	IMRMASK	IMR Mask. A mask ANDed with the predecode bits before using the IMRMATCH field.

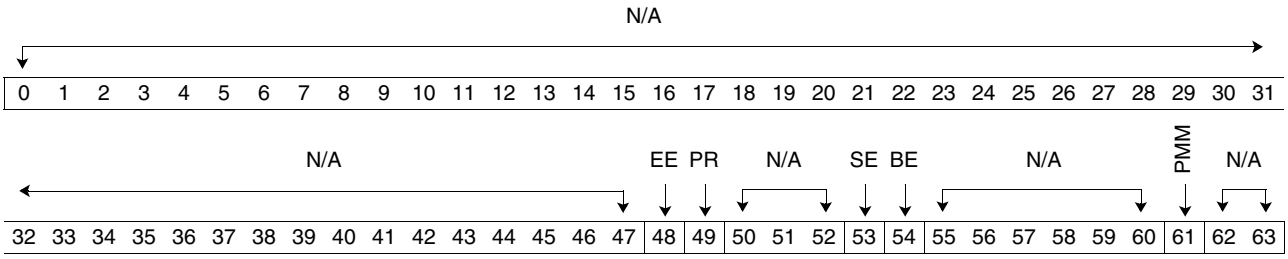
Bits	Field Name	Description
56:59	IMRMATCH	IMR Match. The value that the result of the IMRMASK ANDed with the predecode bits must match to be Stage 2 eligible. All 4 bits of the result must match the IMRMATCH exactly. To match ALL IOPs (that is, the match will always succeed) set IMRSEL equals '0', IMRMASK equals '0000', and IMRMATCH equals '0000'.
60	FCTI	Freeze Counters. 0 The PMCs are incremented. 1 The PMCs are not incremented.
61	—	Reserved
62	FCWAIT	Freeze Counters in Wait State (implies that CNTL[31] equals '0'). 0 The PMCs are incremented. 1 The PMCs (except those counting cycles) are not incremented when CNTL[31] equals '0'.
63	SAMPLE_ ENABLE	0 Sampling is disabled. 1 Sampling is enabled.

10.4.4 Performance Monitor Count Registers PMC1 - 8



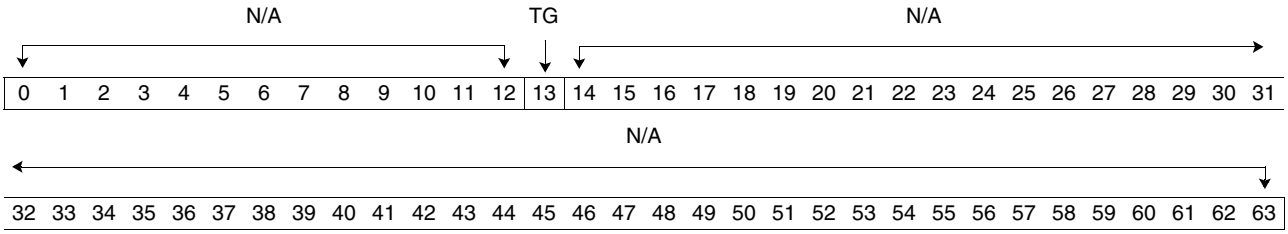
Bits	Field Name	Description
0	CTR_NEG	Counter negative bit.
1:31	CTRDATA	Count data.

10.4.5 Performance Monitor and Trace Related Bits in the Machine State Register (MSR)



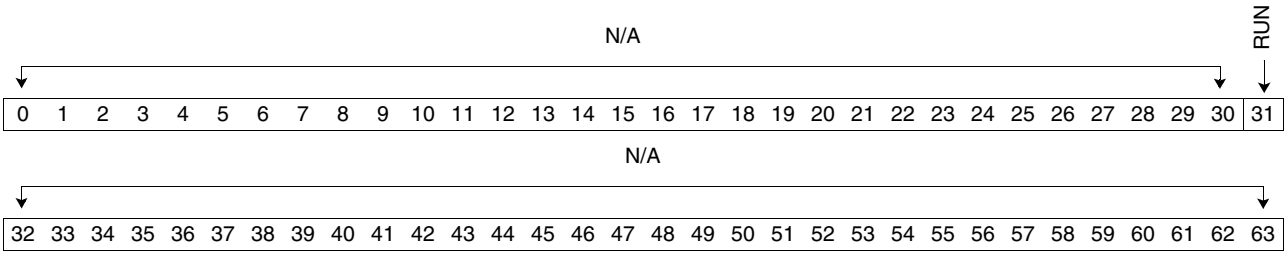
Bits	Field Name	Description
0:47	N/A	Not applicable.
48	EE	External interrupt enable. 0 The processor is disabled for external, decrementer, and performance monitor exceptions. 1 The processor is enabled for external, decrementer, and performance monitor exceptions.
49	PR	Problem (user) state. 0 The processor is privileged to execute any instruction. 1 The processor can execute only non-privileged instructions.
50:52	N/A	Not applicable.
53	SE	Single step trace enable. 0 The processor does not generate a trace exception after instruction completion. 1 The processor generates a trace exception after successfully completing the execution of the next instruction unless that instruction is an Return from Exception Doubleword (**rfid**), which is never traced.
54	BE	Branch trace enable. 0 The processor does not generate a trace exception after branch instruction completion. 1 The processor generates a trace exception after successfully completing the execution of a branch instruction whether the branch is taken.
55:60	N/A	Not applicable.
61	PMM	Performance monitor mode enable. 0 The currently executing process is not marked. 1 The currently executing process is marked. This bit is used to mark a process for the performance monitor. Several performance monitor MMCR0 control bits can then be set to enable counting based on the value of the PMM bit. When an exception occurs, this bit is saved, set to '0' for the duration of the exception processing, and then restored when the **rfid** instruction is executed. If this bit is changed with an **mtmsr** or Move to Machine State Register Doubleword (**mtmsrd**) instruction, the change is not guaranteed to have taken effect until after a subsequent context-synchronizing instruction has completed execution.
62:63	N/A	Not applicable.

10.4.6 Performance Monitor Related Bits in Hardware Implementation-Dependent Register 0 (HID0)



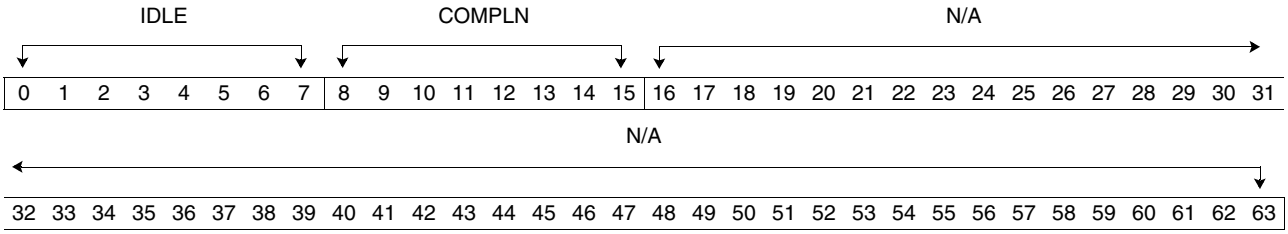
Bits	Field Name	Description
0:12	N/A	Not applicable.
13	TG	Performance monitor threshold granularity. 0 The thresholder counts every processor cycle. 1 The thresholder counts every 32 processor cycles.
14:63	N/A	Not applicable.

10.4.7 Performance Monitor Related Bits in the Control Register (CTRL)



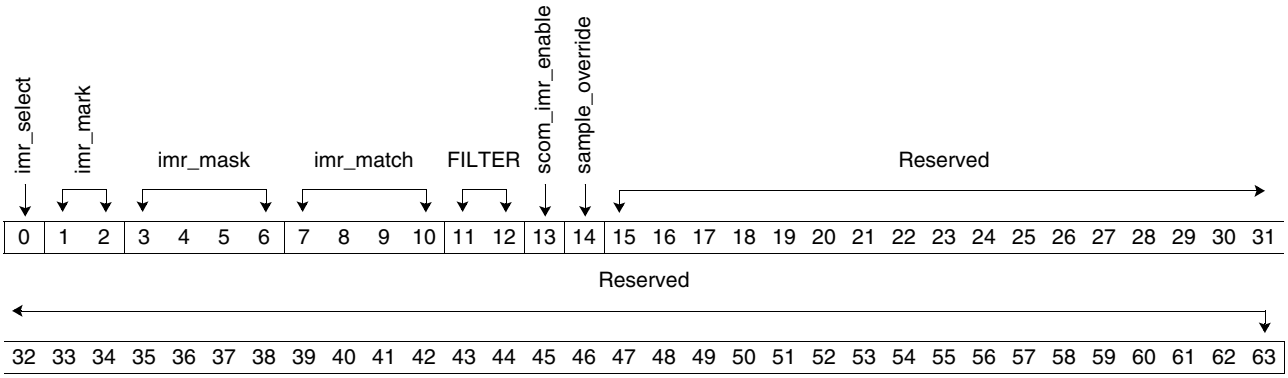
Bits	Field Name	Description
0:30	N/A	Not applicable.
31	RUN	Wait state bit.
32:63	N/A	Not applicable.

10.4.8 Performance Monitor Related Bits in the SCOM0240, 1240 Register (SCOM x'240')



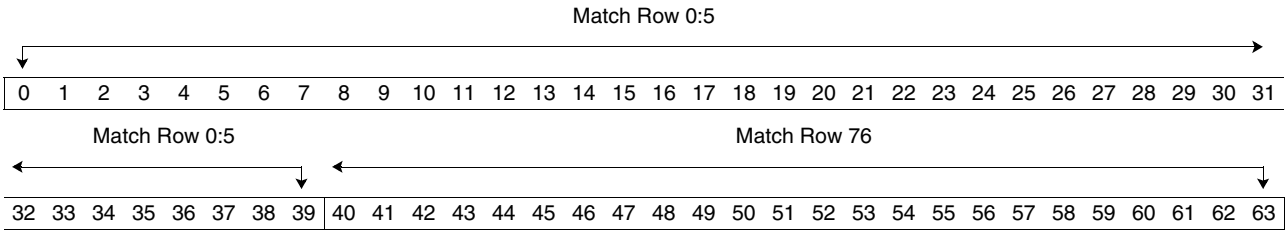
Bits	Field Name	Description
0:7	IDLE	Sampling logic idle delay.
8:15	COMPLN	Sampling logic completion delay.
16:63	N/A	Not applicable.

10.4.9 Performance Monitor Related Bits in the SCOM0360,1360 Register (SCOM x'360')



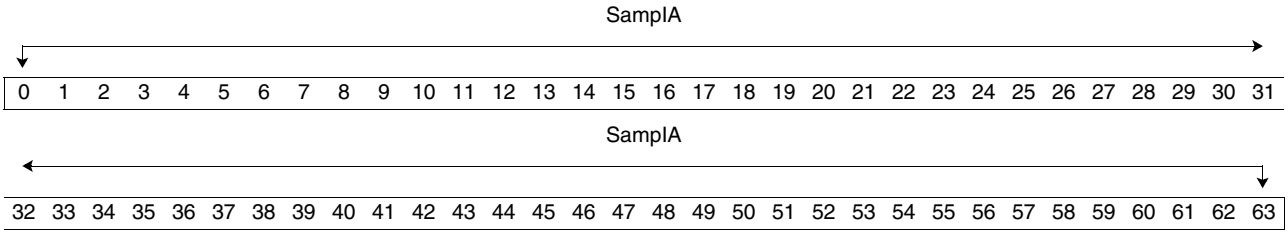
Bits	Field Name	Description
0	imr_select	Same as MMCRA[imr_sel].
1:2	imr_mark	Same as MMCRA[imr_mark] and overrides MMCRA if bit 13 equals '1'.
3:6	imr_mask	Same as MMCRA[imr_mask] and overrides MMCRA if bit 13 equals '1'.
7:10	imr_match	Same as MMCRA[imr_match] and overrides MMCRA if bit 13 equals '1'.
11:12	FILTER	IMR filter random/all and first/all. These two bits form a 2-step filtering operation on the eligible bits associated with the instructions in the group. Bit 11 first determines whether instruction eligibility bits pass the first filter step based on either a random pass/nopass (bit 11 equals '1') choice or an all pass (bit 11 equals '0') choice for each instruction. Bit 12 determines how microcoded instructions are sampled (and has no effect on non-microcoded instructions): 00 No filtering (OR). 01 No filtering (AND). 10 Use Good_Address mode of sampling microcode expansions. 11 Use More_Hits mode of sampling microcode expansions. In Good_Address mode, there is at most one IOP in any microcode expansion that is eligible for sampling. This is (a) the first load/store IOP if there are any load/store IOPs in the expansion, or (b) the first IOP in the final group of the expansion. If the random filter suppresses marking this IOP, then no IOP will be marked for the microcode expansion. In More_Hits mode, multiple IOPs in a microcode expansion are eligible for sampling. These are (a) the first load/store IOP in any group, or (b) the first IOP of the final group. If the random filter suppresses marking the first of these IOPs, a subsequent one might still be sampled. (However, at most one will be marked in a single microcode expansion.)
13	scom_imr_enable	0 Performance monitor fields are used for mark, mask, match. 1 SCOM fields are used for mark, mask, match.
14	sample_override	0 Performance monitor "*ok_to_sample*" indication is used. 1 Overrides performance monitor "*ok_to_sample*" indication.
15:63	—	Reserved.

10.4.10 Performance Monitor Related Bits in the IMC Array (IMC)



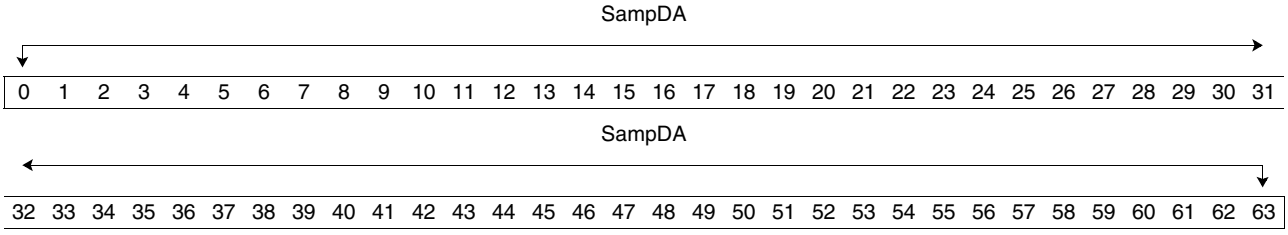
Bits	Field Name	Bit Description
0:39	Match Row 0	Opcode/extended opcode match.
0:39	Match Row 1	Opcode/extended opcode match.
0:39	Match Row 2	Opcode/extended opcode match.
0:39	Match Row 3	Opcode/extended opcode match.
0:39	Match Row 4	Opcode/extended opcode match.
0:39	Match Row 5	Opcode/extended opcode match.
0:63	Match Row 76	Full instruction match.

10.4.11 Performance Monitor Related Bits in the Sampled Instruction Address Register (SIAR)



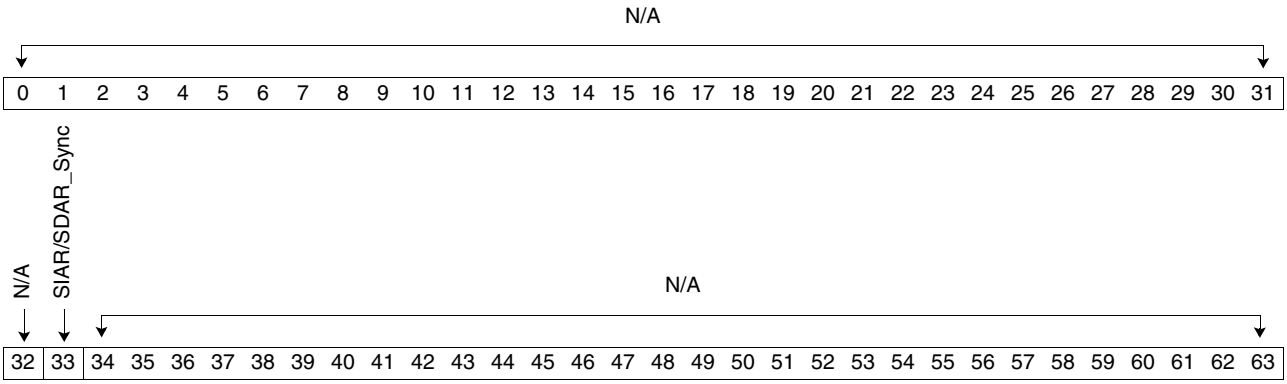
Bits	Field Name	Description
0:63	SampIA	Sampled instruction address.

10.4.12 Performance Monitor Related Bits in the Sampled Data Address Register (SDAR)



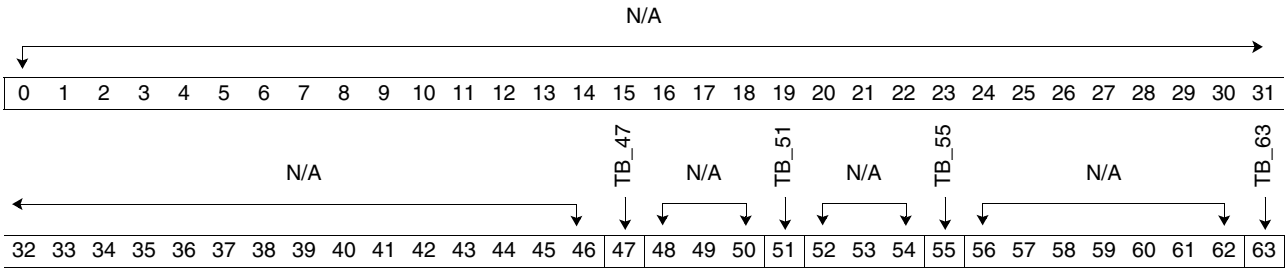
Bits	Field Name	Description
0:63	SampDA	Sampled data address.

10.4.13 Performance Monitor Related Bits in the SRR1 (SRR1)



Bits	Field Name	Description
0:32	N/A	Not applicable.
33	SIAR/SDAR_Sync	SIAR and SDAR contents synchronized.
34:63	N/A	Not applicable.

10.4.14 Performance Monitor Related Bits in the Time-Base Register (TB)



Bits	Field Name	Description
0:46	N/A	Not applicable.
47	TB_47	Time-Base Register bit 47.
48:50	N/A	Not applicable.
51	TB_51	Time-Base Register bit 51.
52:54	N/A	Not applicable.
55	TB_55	Time-Base Register bit 55.
56:62	N/A	Not applicable.
63	TB_63	Time-Base Register bit 63.

10.5 Performance Monitor Event Selection

Event signals are routed from the functional units through the processor performance monitor buses. These signals are multiplexed and divided into byte lanes 0 - 3. A smaller number of events are routed directly to the performance monitor unit (PMU); these events are referred to as direct events. Each PMC can be configured to count a subset of the direct events or one of two possible byte lanes. Counters 1, 2, 5, and 6 can be configured to count events on byte lane 0 or 2, counters 3, 4, 7, and 8 can be configured to count events on byte lane 1 or 3. The selection of event source (direct, byte lane) is controlled by the PMCxSEL field in MMCR1 (where x is the PMC number). *Figure 10-2* shows this selection. *Table 10-2* shows how the PMCxSEL field is used to select which events are monitored.

Performance monitor events fall into three categories:

- Direct: All the information is hardwired to the PMU.
- Bus: All the information is routed over the hierarchical event bus.
- Combined: Some information comes from the event bus; the PMU does additional processing on it.

Figure 10-2. Event Selection

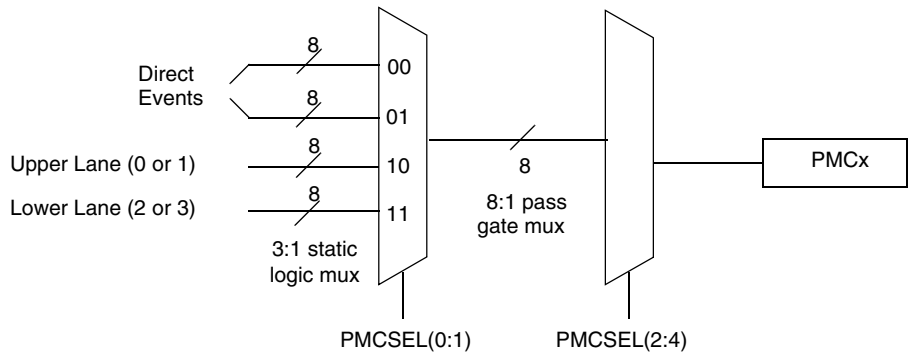


Table 10-2. Performance Monitor Internal Multiplexer PMCxSEL[0:4] Bit Values

PMCSEL[0:1]	PMCSEL[2:4]	Counted Event
00	000-111	Direct Events.
10	000	None. When count_en is '0', turn off counter.
10	111	Cycles.
01	000-111	Direct Events.
10	000-111	Select smaller byte lane.
11	000-111	Select larger byte lane.

10.5.1 Direct Events

As shown in *Table 10-2* on page 317, direct events are selected with PMCxSEL[0:1] set to '0x'. When PMCxSEL[0:1] equals '10' and PMCxSEL[2:4] equals '111', the counter is configured to count cycles. When PMCxSEL[0:1] equals '10' and PMCxSEL[2:4] is '000', the counter is off (counts nothing). The direct events that can be counted are shown in *Table 10-5* on page 320.

Some direct events, such as events that add two other events or interpret the memory source encodes for data or instruction fetches, also require data from the performance monitor events. Although they are listed in *Table 10-5 Direct Events*, they rely on a meaningful configuration of the performance monitor event selections to produce meaningful results.

10.5.1.1 Combined Events

Each PMC can add similar events to produce a single, combined count. For example, each load store unit provides a data cache miss event, which can be added to produce the total data cache miss count. The added events are considered direct events, but they rely on the performance monitor bus being configured properly to produce meaningful results. Because each PMC can receive event signals from two byte lanes on the performance monitor bus, the added events can be configured to add events on one of the two byte lanes. Events cannot be added from different byte lanes. The PMCx_ADDER_SELECT fields in MMCR1 control which byte lanes are used.

10.5.1.2 Source-Encoded Events

Source-encoded events (direct event 7 [PMCxSEL equals '00111'] for data and event 6 [SEL equals '00110'] for instructions) are combined events that count events from a specific source as shown in *Table 10-3* and *Table 10-4* on page 319.

Note: Intervention event sources are only meaningful on multiprocessor systems.

Table 10-3. Event Data Source Encodings

Encoding (0:3)	Event Source
0000	L2 cache
0010	Memory
1000	Shared Intervention (another L2 cache)
1010	Modified Intervention (another L2 cache)
All Others	Reserved

To count data source-encoded events, the performance monitor event bus must be configured as follows:

1. Route LSU1 byte 3 data to the PMU (the "L1 reload data source" LSU1 indirect event) by setting the TD_CP_DBG3SEL field in MMCR1 to '11'.
2. Select the direct event that decodes the required data source. To count L1 data reloads from the L2, for example, PMC1, direct event 7 (the PMC1SEL field in MMCR0 set to '00111') should be used.

Table 10-4. Event Instruction Source Encodings

Encoding (0:3)	Event Source
1001	I-cache
1010	Prefetch buffer
0000	L2 cache
0001	Memory
1111 or 1011	No instructions on bus

To count instruction source-encoded events, the performance monitor event bus should be similarly configured:

1. Route IFU byte 2 data to the PMU (the "iL1 cache data source" IFU indirect event) by setting the TD_CP_DBG2SEL field in MMCR1 to '00' and TTM0SEL to '10'.
2. Select the direct event that decodes the required data source. To count L1 instruction reloads from memory, for example, PMC3, direct event 6 (the PMC3SEL field in MMCR0 set to '00110') should be used.

10.5.1.3 Instruction Counts

Two types of instruction and IOP counting are available with the 970MP performance monitor:

- Direct event 1 (SEL equals '00001') on PMC1, PMC4, PMC6, PMC7, and PMC8 counts instructions according to the IMRMARK field of the MMCRA Register:
  - 00 All stage 1 eligible IOPs
  - 01 Stage 1 eligible IOPs from microcode expansion
  - 10 One IOP per eligible PowerPC instruction
  - 11 First IOP to LSU per eligible PowerPC load/store instruction
- Direct event 9 (SEL equals '01001') on PMC1 - PMC8 always counts PowerPC instructions independent of the IMRMARK field of the MMCRA Register (see *Table 10-5* on page 320).

Table 10-5. Direct Events (Page 1 of 2)

SEL(0:4)	PMC1	PMC2	PMC3	PMC4	PMC5	PMC6	PMC7	PMC8
00 000 plus	Add 0 + 4	Add 1 + 5	Add 2 + 6	Add 3 + 7	Add 3 + 7	Add 2 + 6	Add 1 + 5	Add 0 + 4
MMCR1(24:31) = '0', '1'	MMCR1[24] = '0' byte lane 0	MMCR1[25] = '0' byte lane 0	MMCR1[30] = '0' byte lane 1	MMCR1[31] = '0' byte lane 1	MMCR1[27] = '0' byte lane 0	MMCR1[26] = '0' byte lane 0	MMCR1[29] = '0' byte lane 1	MMCR1[28] = '0' byte lane 1
	MMCR1[24] = '1' byte lane 2	MMCR1[25] = '1' byte lane 2	MMCR1[30] = '1' byte lane 3	MMCR1[31] = '1' byte lane 3	MMCR1[27] = '1' byte lane 2	MMCR1[26] = '1' byte lane 2	MMCR1[29] = '1' byte lane 3	MMCR1[28] = '1' byte lane 3
00 001	number of instructions complete	work held	stop completion	number of instructions complete	dispatch_success	number of instructions complete	number of instructions complete	number of instructions complete
00 010	marked group dispatch	LSU empty (load miss queue [LMQ] and store reorder queue [SRQ] empty)	LSU empty (LMQ and SRQ empty)	Fixed-point unit 0 (FXU0) idle and FXU1 busy	FXU0 idle and FXU1 idle	FXU0 busy and FXU1 busy	FXU0 busy and FXU1 idle	external interrupt
00 011	marked store complete	threshold timeout event	marked store with interrupt complete	SRQ empty	one or more PowerPC instructions completed	marked store sent to STS	group completed	group dispatch reject
00 100	global completion table (GCT) empty	group dispatch	cycles in supervisor mode	marked group complete	group marked in IDU	FXU marked instruction finish	FPU marked instruction finish	LSU marked instruction finish
00 101	run_cycles; that is, # cycles when CNTL[31] = '1'	branch unit (BRU) marked instruction finish	VPU marked instruction finish	condition register unit (CRU) marked instruction finish	marked group complete time out	marked group issued	marked instruction finish any unit	time base event
00 110	Instruction source encode 0000	Instruction source encode 0001	Instruction source encode 0010	Instruction source encode 0011	Instruction source encode 0100	Instruction source encode 0101	Instruction source encode 0110	Instruction source encode 0111
00 111	Data source encode 0000	Data source encode 0001	Data source encode 0010	Data source encode 0011	Data source encode 0100	Data source encode 0101	Data source encode 0110	Data source encode 0111
01 000	Counter OFF	Counter OFF	Counter OFF	Counter OFF	Counter OFF	Counter OFF	Counter OFF	Counter OFF
01 001	number of instructions complete	number of instructions complete	number of instructions complete	number of instructions complete	number of instructions complete	number of instructions complete)	number of instructions complete	number of instructions complete
01 010	Overflow from counter 8	Overflow from counter 1	Overflow from counter 2	Overflow from counter 3	Overflow from counter 4	Overflow from counter 5	Overflow from counter 6	Overflow from counter 7
01 011	reserved	GCT empty by SRQ full	reserved	reserved	—/A1a/A2a/A3a (*1) (See *Table 10-6* on page 323)	reserved	—/A1b/A2b/A3b (*1) (See *Table 10-6* on page 323)	reserved

Table 10-5. Direct Events (Page 2 of 2)

SEL(0:4)	PMC1	PMC2	PMC3	PMC4	PMC5	PMC6	PMC7	PMC8
01 100	reserved	reserved	reserved	reserved	—/A1c/A2c/— (*1) (See *Table 10-6* on page 323)	reserved	—/A1d/A2d/—(*1) (See *Table 10-6* on page 323)	reserved
01 101	Instruction source decode 1000	Instruction source encode 1001	Instruction source encode 1010	Instruction source encode 1011	Instruction source encode 1100	Instruction source encode 1101	Instruction source encode 1110	Instruction source encode 1111
01 110	Byte 3 decode 1000	Data source encode 1001	Data source encode 1010	Data source encode 1011	Data source encode 1100	Data source encode 1101	Data source encode 1110	Data source encode 1111
01 111	Cycles	Cycles	Cycles	Cycles	Cycles	Cycles	Cycles	Cycles

10.5.2 Over 32-Bit Count

The 970MP PMU can chain together multiple 32-bit PMCs to create up to a 256-bit wide PMC Register when used in conjunction with overflow counting. This is useful for performance measurement on high clock-rate machines. The maximum count value depends on the following settings:

- PMCn can count over 32-bits when PMC*n+1*SEL(0:4)(where *n* is 1 - 7) is set to '01010'.
- PMC8 can count over 32-bits when PMC1SEL(0:4) is set to '01010'.
- When PMC*n+1* uses this overflow counting function, PMCn is prohibited from asserting an exception signal when a negative condition occurs (PMC1CE(PMCjCE) equals '1' and PMCn[0] is '1').

10.5.2.1 Examples of Over Bit Count

Example 1

When PMC1 is set to '00100' (GCT empty) and PMC2 is set to '01010' (overflow function), then PMC2 works as the upper 32 bits of PMC1. In this case, the overflow exception is only asserted by PMC2 (never by PMC1) when PMCjCE equals '1' (don't care PMC1CE) and PMC2[0] is '1'.

Example 2

When PMC8 is set to '00001' (number of instructions complete) and PMC1 are set to '01010' (overflow function), then PMC1 functions as the upper 32 bits of PMC8. In this case, the overflow exception is only asserted by PMC1 (never by PMC8) when PMC1CE equals '1' (don't care PMCjCE) and PMC1[0] is '1'.

Example 3

When PMC1 is set to '00100' (GCT empty) and PMC2, PMC3, and PMC4 is set to '01010' (overflow function), then PMC4, PMC3, and PMC2 function as the upper 96 bits of PMC1. In this case, the overflow exception is only asserted by PMC4 (never by PMC1, PMC2, or PMC3) when PMCjCE equals '1' (don't care PMC1CE) and PMC4[0] is '1'.

10.5.3 Speculative Count

PMC5 and PMC7 support the speculative count function with a backup register. This is enabled when MMCR1[62:63] is set to '01', '10', or '11' and a speculative event is selected (PMC[5,7]SEL equals '01011' or '01100'). The PMC starts counting speculatively whenever a next-to-complete (NTC) group completion stops (or GCT empty happens). The PMC then stores the counts to itself and its backup register if the last finished event matches what the PMC initially set up. If there is no match, the PMC restores the old count value from the backup register. This allows the PMU to establish a cycles per instruction (CPI) breakdown for various categories (CPI contribution due to an instruction-cache [I-cache] miss, data cache [D-cache] miss, LSU, FXU, FPU, and so on).

A negative condition exception only occurs when the count value is not speculative and a negative condition occurs. (When PMC1CE[PMCjCE] is set to '1' and the backup register's negative bit is '1'.)

Table 10-6 on page 323 lists the speculative count events.

Table 10-6. Speculative Count Events

PMC Number		SEL(0:4)	MMCR1 Condition		Count Events	See Note
			Bit 62	Bit 63		
	5, 7	01011	0	0	Reserved	
A1a	5	01011	0	1	Completion stall by LSU instruction	
A2a	5	01011	1	0	Completion stall by FXU instruction	
A3a	5	01011	1	1	Completion stall by D-cache miss	
A1b	7	01011	0	1	Completion stall by FPU instruction	
A2b	7	01011	1	0	Completion stall by FXU long instruction	
A3b	7	01011	1	1	Completion stall by reject	
	5, 7	01100	0	0	Reserved	
A1c	5	01100	0	1	Completion stall by FPU long instruction	
A2c	5	01100	1	0	GCT empty by I-cache miss	1
A1d	7	01100	0	1	Completion stall by reject (ERAT miss)	
A2d	7	01100	1	0	GCT empty by branch miss predict	1
	5, 7	01100	1	1	Reserved	

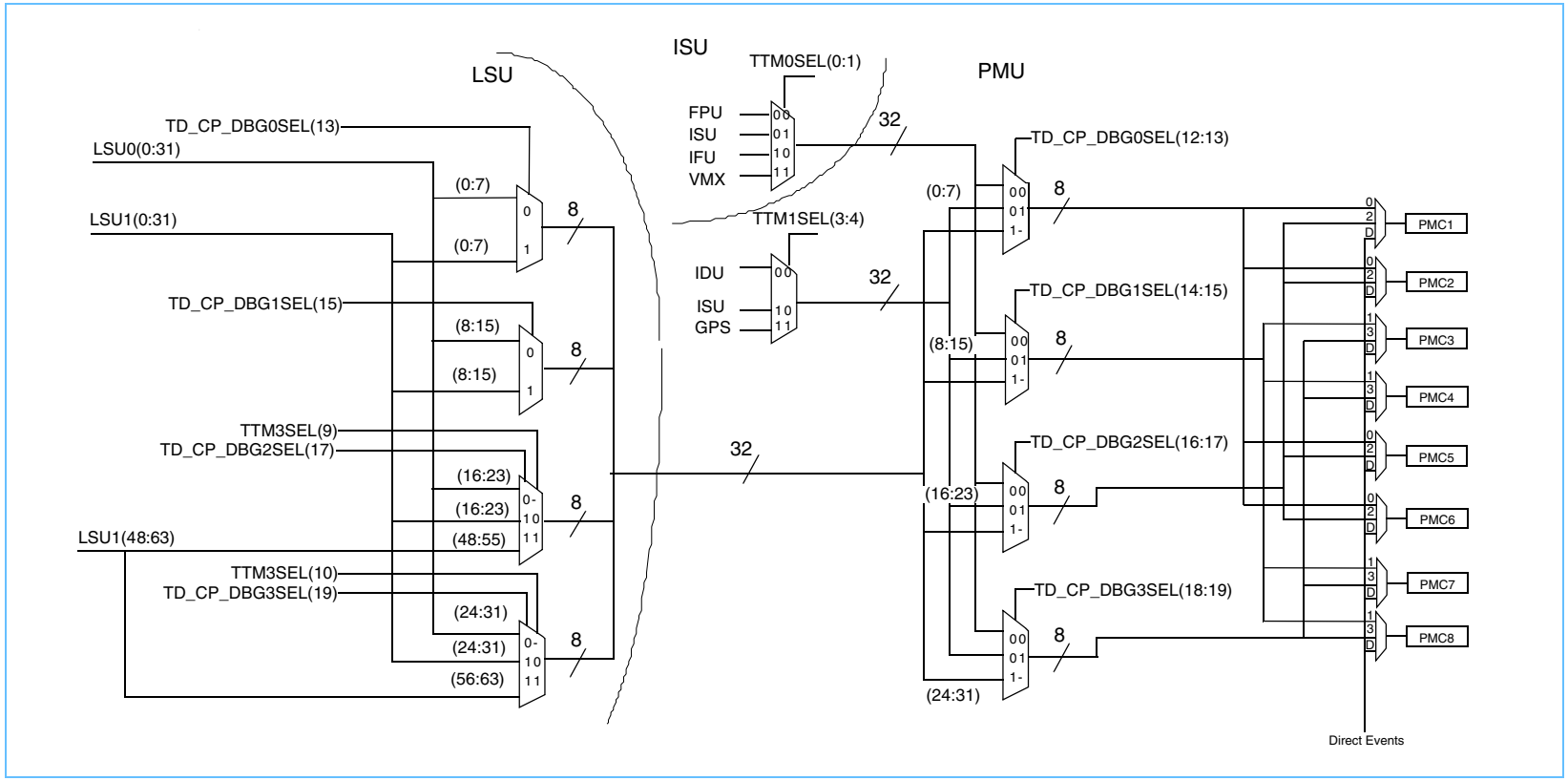
1. This count event also requires MMCR1 bits. They should be set as follows:
   Bit 1 = '1' and bits 0, 16, and 17 = '0' or
   Bits 3 and 17 = '1' and bits 4 and 16 = '0'

Speculative events are also able to count in the over 32-bit count mode. In this case, the overflow value is carried to the upper PMC only when the counting value is not speculative. For this reason, the upper PMC does not require a backup register to copy and restore the count value.

10.6 Configuring the Performance Monitor Bus

The 970MP performance monitor bus is configured through a series of hierarchal multiplexers, as shown in *Figure 10-3* on page 324. This diagram also shows that all unit event buses are 32 bits, except for the LSU1 that sources an extra 16 bits, denoted as LSU1[48:63]. The LSU0 and LSU1 event buses are multiplexed into a single 32-bit LSU event bus, using the multiplexers shown at the left of *Figure 10-3*. Basically, the multiplexers, on a byte basis, select either the LSU0 or the LSU1 events. The extra LSU1 events are selected using MMCR1[9:10], the TTM3SEL select signals.

Figure 10-3. 970MP Performance Monitor Bus Configuration



The rest of the first level of selection is controlled by the TTM0 and TTM1 multiplexers. These control from which unit the non-LSU event signals are selected. The ISU can be selected by more than one multiplexer (TTM0 and TTM1). The TTM multiplexers are controlled by the TTMxSEL fields in MMCR1. MMCR1[0:1] control TTM0 and MMCR1[3:4] control TTM1.

The three 32-bit outputs of the LSU, TTM0, and TTM1 multiplexers are sent to the TM_DEBUG multiplexers, which are controlled by the 2-bit TD_CP_DBGxSEL fields in the MMCR1; bits 12:13 control TM_DEBUG0, bits 14:15 control TM_DEBUG1, bits 16:17 control TM_DEBUG2, and bits 18:19 control TM_DEBUG3.

After the performance monitor bus is configured, individual events can be selected, as described at the beginning of this section. *Table 10-7* shows the events available through the performance monitor bus and the TTM and TM_DEBUG multiplexer used to select them.

Table 10-7. Performance Monitor Bus Assignments (Page 1 of 8)

	Bits [0:31]	Byte Lane	Bits [0:7]	Event Description
FPU: TTM0 = '00', TD_CP_DBGxSEL = '00'				
	0	0	0	FPU0 divide
	1	0	1	FPU0 mult-add
	2	0	2	FPU0 square root
	3	0	3	FPU0 add, mult, sub, compare, fsel
	4	0	4	FPU1 divide
	5	0	5	FPU1 mult-add
	6	0	6	FPU1 square root
	7	0	7	FPU1 add, mult, sub, compare, fsel
	8	1	0	FPU0 move, estimate
	9	1	1	FPU0 round, convert
	10	1	2	FPU0 estimate
	11	1	3	FPU0 finished and produced a result
	12	1	4	FPU1 move, estimate
	13	1	5	FPU1 round, convert
	14	1	6	FPU1 estimate
	15	1	7	FPU1 finished and produced a result

Table 10-7. Performance Monitor Bus Assignments (Page 2 of 8)

	Bits [0:31]	Byte Lane	Bits [0:7]	Event Description
	16	2	0	FPU0 denormalized operand
	17	2	1	FPU0 stall 3
	18	2	2	FPU0 store
	19	2	3	FPU0 single precision
	20	2	4	FPU1 denormalized operand
	21	2	5	FPU1 stall 3
	22	2	6	FPU1 store
	23	2	7	FPU1 single precision
	24	3	0	Floating-Point Status and Control Register (FPSCR)
	25	3	1	FPU0 multiply
	26	3	2	FPU0 compare
	27	3	3	FPU0 select
	28	3	4	FPU1 multiply
	29	3	5	FPU1 compare
	30	3	6	FPU1 select
	31	3	7	Floating-point stall store
IFU: TTM0 = '10', TD_CP_DBGxSEL = '00'				
	0:15	0:1	0:7	Nothing
	16:19	2	0:3	L1 I-cache data source
	20	2	4	Valid instruction available
	21	2	5	Cycles IFU is held by pipeline hold
	22	2	6	Instruction prefetch installed in prefetch buffer
	23	2	7	L2 prefetch request
	24	3	0	I-ERAT write
	25	3	1	Branch execution issue valid
	26	3	2	Branch miss predict due to Condition Register (CR) value
	27	3	3	Branch miss predict due to target address predict
	28	3	4	Cycles L1 I-cache write active
	29:31	3	5:7	Nothing

Table 10-7. Performance Monitor Bus Assignments (Page 3 of 8)

	Bits [0:31]	Byte Lane	Bits [0:7]	Event Description
IDU: TTM1 = '00', TD_CP_DBGxSEL = '01'				
	0	0	0	Instruction queue has three slots full
	1	0	1	Instruction queue has one slot full
	2	0	2	Instruction queue has six slots full
	3	0	3	Instruction queue has zero slots full
	4	0	4	Instruction queue has four slots full
	5	0	5	Instruction queue has two slots full
	6	0	6	Instruction queue has seven slots full
	7	0	7	Instruction queue has eight slots full
	8	1	0	Instruction queue has five slots full
	9:15	1	1:7	Instruction queue full
	16:31	2:3	0:7	Nothing
LSU0: (See *Figure 10-3 970MP Performance Monitor Bus Configuration* on page 324) TD_CP_DBGxSEL = '1x'				
	0	0	0	Instruction translation lookaside buffer (TLB) miss
	1	0	1	Instruction segment lookaside buffer (SLB) miss
	2	0	2	Data ERAT (D-ERAT) miss side 0
	3	0	3	Snoop TLB Invalidate Entry (**tlbie**)
	4	0	4	Data TLB miss
	5	0	5	Data SLB miss
	6	0	6	D-ERAT miss side 1
	7	0	7	Tablewalk duration
	8	1	0	Marked flush unaligned load side 0
	9	1	1	Marked flush unaligned store side 0
	10	1	2	Marked flush from load reorder queue (LRQ) store-hit-load (SHL), load-hit-load (LHL) side 0
	11	1	3	Marked flush SRQ load-hit-store (LHS) side 0
	12	1	4	Marked flush unaligned load side 1
	13	1	5	Marked flush unaligned store side 1
	14	1	6	Marked flush from LRQ store-hit-load (shl), load-hit-load (lhl) side 1
	15	1	7	Marked flush SRQ LHS side 1

Table 10-7. Performance Monitor Bus Assignments (Page 4 of 8)

	Bits [0:31]	Byte Lane	Bits [0:7]	Event Description
	16	2	0	Marked L1 D-cache load miss side 0
	17	2	1	Store conditional (**stcx**) failed
	18	2	2	Marked IMR reload
	19	2	3	Marked L1 D-cache store miss
	20	2	4	Marked L1 D-cache load miss side 1
	21	2	5	**stcx** passed
	22	2	6	Marked **stcx** fail
	23	2	7	Load and reserve indexed (**larx**) executed 0
	24	3	0	Floating-point load side 0
	25	3	1	L1 cache prefetch request
	26	3	2	Out of streams
	27	3	3	L2 cache prefetch
	28	3	4	Floating-point load side 1
	29	3	5	VPU type L2 prefetch
	30	3	6	Data stream touch (**dst**) stream start
	31	3	7	New stream allocated
LSU1: (See *Figure 10-3 970MP Performance Monitor Bus Configuration* on page 324) TD_CP_DBGxSEL = '1x'				
	0	0	0	Flush unaligned load side 0
	1	0	1	Flush unaligned store side 0
	2	0	2	Flush from LRQ SHL, LHL side 0
	3	0	3	Flush SRQ LHS side 0
	4	0	4	Flush unaligned load side 1
	5	0	5	Flush unaligned store side 1
	6	0	6	Flush from LRQ SHL, LHL side 1
	7	0	7	Flush SRQ LHS side 1
	8	1	0	L1 D-cache load side 0
	9	1	1	L1 D-cache store side 0
	10	1	2	L1 D-cache load miss side 0
	11	1	3	L1 D-cache store miss
	12	1	4	L1 D-cache load side 1
	13	1	5	L1 D-cache store side 1
	14	1	6	L1 D-cache load miss side 1
	15	1	7	L1 D-cache entries invalidated from L2

Table 10-7. Performance Monitor Bus Assignments (Page 5 of 8)

	Bits [0:31]	Byte Lane	Bits [0:7]	Event Description
	16	2	0	SRQ store forwarding side 0
	17	2	1	SRQ slot 0 valid
	18	2	2	LRQ slot 0 valid
	19	2	3	LSU0 reject
	20	2	4	SRQ store forwarding side 1
	21	2	5	SRQ slot 0 allocated
	22	2	6	LRQ slot 0 allocated
	23	2	7	LSU1 reject
	24:27	3	0:3	L1 cache reload data source
	28	3	4	L1 cache reload data valid
	29	3	5	LMQ slot 0 valid
	30	3	6	LMQ slot 0 allocated
	31	3	7	LMQ full
	32:47	0:1	0:7	Nothing
	48	2	0	SRQ reject 0 - load hit store
	49	2	1	LMQ reject 0 - LMQ full or missed data coming
	50	2	2	LSU0 reject - reload critical data forward (CDF) or tag update collision
	51	2	3	LSU0 reject - ERAT miss
	52	2	4	SRQ reject 1- load hit store
	53	2	5	LMQ reject 1- LMQ full or missed data coming
	54	2	6	LSU1 reject - reload CDF or tag update collision
	55	2	7	LSU1 reject - ERAT miss
	56:58	3	0:3	L1 cache reload data source
	60	3	4	Marked L1 cache reload data source valid
	61	3	5	LMQ load-hit-reload merge
	62	3	6	Marked SRQ valid
	63	3	7	Nothing

Table 10-7. Performance Monitor Bus Assignments (Page 6 of 8)

	Bits [0:31]	Byte Lane	Bits [0:7]	Event Description
ISU: TTM0 = '01', TD_CP_DBGxSEL = '00' TTM1 = '10', TD_CP_DBGxSEL = '01'				
	0	0	0	Completion table full
	1	0	1	Floating-Point Register (FPR) mapper full
	2	0	2	Integer Exception Register (XER) mapper full
	3	0	3	FPU0 issue queue full
	4	0	4	CR mapper full
	5	0	5	Branch (BR) issue queue full
	6	0	6	Link Register/ Counter Register (LR/CTR) mapper full
	7	0	7	FPU1 issue queue full
	8	1	0	FXU0/LSU0 issue queue full
	9	1	1	CR issue queue full
	10	1	2	LRQ full
	11	1	3	SRQ full
	12	1	4	FXU1/LSU1 issue queue full
	13	1	5	Flush originated by LSU
	14	1	6	Flush originated by branch miss predict
	15	1	7	Flush (includes LSU, branch miss predict)
	16:18	2	0:2	Instructions dispatched count
	19	2	3	Dispatch valid
	20	2	4	Dispatch reject
	21	2	5	Nothing
	22	2	6	Group experienced a branch redirect
	23	2	7	Group experienced a branch miss predict
	24	3	0	Nothing
	25	3	1	Dispatch blocked by scoreboard
	26	3	2	FXU0 produced a result
	27	3	3	Duration of the external interrupt enable in the Machine State Register (MSR[EE] = '0')
	28	3	4	Nothing
	29	3	5	General Purpose Register (GPR) mapper full
	30	3	6	FXU1 produced a result
	31	3	7	MSR(EE) equals '0' and interrupt pending

Table 10-7. Performance Monitor Bus Assignments (Page 7 of 8)

	Bits [0:31]	Byte Lane	Bits [0:7]	Event Description
Vector: TTM0 = '11', TD_CP_DBGxSEL = '01'				
	0	0	0	Arithmetic logic unit (ALU) issue queue full
	1	0	1	Vector permute (VPERM) issue queue full
	2	0	2	ALU issue marked instruction
	3	0	3	VPERM issue marked instruction
	4	0	4	Saturation zero to one
	5	0	5	VPU mapper full
	6	0	6	Store issue marked instruction
	7	0	7	Nothing
	8:15 below selected by OVMA.USADEC2.CHICKEN1.IO.L2(7) = '0' (default)			
	8	1	0	Finish with IMR
	9	1	1	Generic forward
	10	1	2	Vector ALU issue count
	11	1	3	Denormalized traps
	12	1	4	Saturation bit set
	13	1	5	Finish contention cycle
	14	1	6	Nothing
	15	1	7	Nothing
	16:19 below selected by OVMP.RPRPM.MODE_PMON_MISC.IO.L2 = '0' (default)			
	16	2	0	Instruction finish with IMR
	17	2	1	Forwarding occurred from VPERM or ALU or load
	17	2	2	Issue valid
	19	2	3	Saturation count for valid instruction
STS: TTM1 = '11', TD_CP_DBGxSEL = '01'				
	0	0	0	L2 cache access collision with L2 prefetch (Data Stream Touch [DST])
	1	0	1	L2 cache access collision with L2 prefetch (non-DST)
	2	0	2	L2 cache access for store
	3	0	3	L2 cache miss on store access (recent [R], shared [S], or invalid [I])
	4	0	4	L2 cache miss; bus response is modified intervention
	5	0	5	L2 cache miss; bus response is shared intervention
	6	0	6	I = '1' store operation (before gathering)
	7	0	7	I = '1' store operation completed on bus

Table 10-7. Performance Monitor Bus Assignments (Page 8 of 8)

	Bits [0:31]	Byte Lane	Bits [0:7]	Event Description
	8	1	0	I = '1' load operation completed on bus
	9	1	1	Cacheable store operation (before gathering)
	10	1	2	Master bus transactions completed
	11	1	3	Master bus transactions retried
	12	1	4	Master L2 cache store transaction on bus was retried
	13	1	5	Master L2 cache read transaction on bus was retried
	14	1	6	Master SYNC operation competed
	15	1	7	Master SYNC operation retried
	16	2	0	Load or store dispatch retries due to castout (CO) conflicts
	17	2	1	Load or store dispatch retries due to snoop conflicts
	17	2	2	Load or store dispatch retries
	19	2	3	All read/claim state machines busy
	20	2	4	All CO state machines busy
	21	2	5	All snoop state machines busy
	22	2	6	Cacheable store queue full
	23	2	7	I = '1' store queue full
	24	3	0	Snoop (external)
	25	3	1	Snoop state machine dispatched
	26	3	2	Snoop retried due to any conflict
	27	3	3	Snoop retried due to all snoop state machines busy
	28	3	4	Snoop caused cache transition from modified (M) to exclusive (E) or shared (S)
	29	3	5	Snoop caused cache transition from E to S
	30	3	6	Snoop caused cache transition from E or S to recent (R) or invalid (I)
	31	3	7	Snoop caused cache transition from M to I

10.7 Enabling the Performance Monitor Counters

10.7.1 Machine States

The eight counters in the 970MP performance monitor can be enabled to count events as a result of a number of machine state conditions and triggering events. The machine state conditions and triggering events are enabled by the settings of the MMCR0, MMCR1, and MMCRA Register control fields, combined with the values of the performance monitor-related bits in the MSR and other SPRs. While machine states and triggering events are closely related in their effect on performance monitor behavior, it is easier to understand them if the two are first considered separately, as described in this section for the machine states and in *Section 10.7.2* on page 334 for the triggering events.

The term machine state condition as it is used here includes:

- Supervisor versus user (problem) state (MSR[PR], MMCR0[FCS, FCP])
- Marked versus unmarked process state (MSR[PMM], MMCR0[FCM1, FCM0])
- Conditional versus unconditional counting state (MMCR0[FC], MMCRA[FC1:4], FC[5:8], CTRL[31], MMCRA[FCWAIT])
- Wait state versus non-wait state (CTRL[31], MMCRA[FCWAIT])

By combining the state of the machine with the events selected for counting, many different aspects of performance can be obtained for a given program.

For example, a programmer might want to gather statistics on only a particular process. This can be done by doing the following steps:

1. Set the appropriate bits in MMCR0 that enable counting only for a marked process.
2. Before each run of the selected process begins, set the performance monitor mode bit in the Machine State Register (MSR[PMM]) to the value that marks that process.
3. After each run of the selected process ends, set the performance monitor mode bit to the value that unmarks that process.

Another example follows:

The performance monitor can be set up to count only when the machine is in the supervisor state by ensuring that the MMCR0 bits that specify counting are enabled only when the machine is in the supervisor (privileged) state and are disabled when the machine is in the user (problem) state.

Table 10-8 on page 334 illustrates several different counting scenarios.

Table 10-8. Examples of Event Counter Enabling States

Counting State	MMCR0[Bit] = Value	MSR[Bit] = Value
Disable all counting	FC = '1'	Does not count for all values of PR, PMM
Enable all counting	FC = '0'	Counts[1] for all values of PR, PMM
Enable counting in supervisor state only	FCP = '1', FCS = '0'	Counts when PR = '0'
Disable counting in supervisor state only	FCS = '1', FCP = '0'	Counts when PR = '1'
Enable counting in user (problem) state only	FCS = '1', FCP = '0'	Counts when PR = '1'
Disable counting in user (problem) state only	FCS = '0', FCP = '1'	Counts when PR = '0'
Enable counting for marked processes only	FCM0 = '1', FCM1 = '0'	Counts when PMM = '1'
Disable counting for marked processes only	FCM0 = '0', FCM1 = '1'	Does not count when PMM = '1'
Enable counting for unmarked processes only	FCM0 = '0', FCM1 = '1'	Counts when PMM = '0'
Disable counting for unmarked processes only	FCM0 = '1', FCM1 = '0'	Does not count when PMM = '0'
Enable counting for marked processes in supervisor state only	FCP = '1', FCS = '0', FCM0 = '1', FCM1 = '0'	Counts when PR = '0' and PMM = '1'
Enable counting for unmarked processes in supervisor state only	FCP = '1', FCS = '0', FCM0 = '0', FCM1 = '1'	Counts when PR = '0' and PMM = '0'
Enable counting for marked processes in user (problem) state only	FCP ='0', FCS = '1', FCM0 = '1', FCM1 = '0'	Counts when PR = '1' and PMM = '1'
Enable counting for unmarked processes in user (problem) state only	FCP = '0', FCS = '1', FCM0 = '0', FCM1 = '1'	Counts when PR = '1' and PMM = '0'

Note:

1. All enables are based on whether the other MMCRx and/or MSR bits permit this action.

10.7.2 Trigger Events

Machine states that determine counter activity have been presented in *Section 10.7.1* on page 333. Several examples of states and their corresponding counter behaviors were shown in order to illustrate some of the more common uses. In addition to counting enable/disable for various machine states, there are certain kinds of performance monitor conditions and events that can affect performance monitor counting activity. The occurrence of these conditions and events, which together are called trigger events, combined with the controls that enable the trigger events, can be used together with machine states to further control performance monitor activity.

The term trigger event as it is used for the 970MP performance monitor includes the following conditions:

- The time-base transition event can occur when a selected time-base bit changes from '0' to '1'. The time-base event setup uses the following fields: TB_REG[47, 51, 55, 63], HID0[13], MMCR0[TBSEL]. The time-base event enable uses the following field: MMCR0[TBEE]. The possibility of side effects when an enabled time-base event occurs uses the following fields: MMCR0[FCECE, TRIGGER].
- The counter negative condition for PMC1 can occur when the value in PMC1 is negative. The PMC1 counter negative condition setup uses the following field: PMC1[0]. The PMC1 counter negative condition enable uses the following field: MMCR0[PMC1CE]. The possibility of side effects when the PMC1 counter negative condition occurs uses the following fields: MMCR0[FCECE,TRIGGER].
- The counter negative condition for PMCj ($2 \leq j \leq 8$) occurs when the value in any PMCj is negative. The PMCj counter negative condition setup uses the following field: PMCj[0]. The PMCj counter negative con-

dition enable uses the following field: MMCR0[PMCjCE]. The possibility of side effects when the PMCj counter negative condition occurs uses the following fields: MMCR0[FCECE, TRIGGER].

Note: The three kinds of trigger events can occur independently of each other and independently of whether the condition or event is enabled. For example, a counter can go negative regardless of whether the counter negative condition for that counter is enabled. However, the side effects of that counter going negative will be seen only if the counter goes negative, the counter negative condition for that counter is enabled, and the side effects are also enabled.

By combining the trigger events and their respective enables with the time-related values obtained in the counters, performance profiles of different kinds of events can be obtained for a given program.

10.7.2.1 Time-Base Transition Events

Time-base events occur when the selected time-base bit (TB_REG[47, 51, 55, 63], HID0[13], MMCR0[TBSEL]) changes value from '0' to '1'. If the time-base transition event is enabled (MMCR0[TBEE]), then any performance monitor action that is started by the occurrence of a trigger event (MMCR0[TRIGGER]) will be initiated. Any performance monitor action that is stopped by the occurrence of a trigger event (MMCR0[FCECE]) will be terminated. In multiprocessor systems with the Time-Base Registers synchronized among the processors, time-base transition events can be used to correlate the performance monitor data obtained by the several processors provided that software has specified the same TBSEL value for all of the processors in the system.

The frequency of the time base is implementation dependent, and a system service routine should be invoked to obtain the frequency before a value for TBSEL is chosen.

10.7.2.2 PMC1 Counter Negative Condition Events

The PMC1 counter negative condition occurs when PMC1[0] equals '1', which can occur either through counting from '0', counting from a positive value greater than '0', or through setting the PMC1[0] bit to '1' in an interrupt or service routine. If the PMC1 negative count condition is enabled (MMCR0[PMC1CE]), any performance monitor action that is started by the occurrence of a trigger event (MMCR0[TRIGGER]) will be initiated, and any performance monitor action that is stopped by the occurrence of a trigger event (MMCR0[FCECE]) will be terminated when PMC1[0] becomes negative.

For example, if the PMC1 negative count condition is to be used to start the other PMCj counters after a designated number of cycles has elapsed, the set up would be as follows:

1. PMC1 is set to the value (x'8000 0000' - <number of cycles>).
2. PMC1SEL is set up to count cycles.
3. MMCR0[PMC1CE] is set to enable the PMC1 negative counter condition.
4. TRIGGER is enabled.

In this case, it is not necessary to enable the PMC1 counter negative condition because the TRIGGER uses either PMC1 negative or an enabled trigger event to start the enabled PMCjs counting.

10.7.2.3 PMCj (2 ≤ j ≤ 8) Counter Negative Condition Events

The PMCj (2 ≤ j ≤ 8) counter negative condition event occurs when PMCj[0] equals '1' (2 ≤ j ≤ 8), which can occur through counting from '0', counting from a positive value greater than '0', or through setting the PMCj[0] (2 ≤ j ≤ 8) bit to '1' in an interrupt or service routine. If the PMCj (2 ≤ j ≤ 8) counter negative condition event is enabled (MMCR0[PMCjCE]), any performance monitor action that is started by the occurrence of a trigger event (MMCR0[TRIGGER]) will be initiated, and any performance monitor action that is stopped by the occurrence of a trigger event (MMCR0[FCECE]) will be terminated, when any of the PMCj (2 ≤ j ≤ 8) counters become negative.

10.7.3 Method for Enabling or Disabling Performance Monitor Counting

This section describes the fundamental mechanism that should be used to place the selected values into the Performance Monitor Registers and other SPRs to initiate and terminate counting.

Once all of the control and event selection choices have been made, there are 32-bit and 64-bit values that must be placed into each of the registers associated with performance monitor counting. These values are placed in the registers with the **mtspr** instruction, which may be executed only in supervisor mode.

Note: If the Performance Monitor Counter Register values are changed while the performance monitor is enabled for counting, then the resulting performance monitor state is undefined.

The basic steps for enabling the performance monitor counting activity are as follows:

1. Enter supervisor mode.
2. Execute a synchronizing instruction.
3. Execute all **mtspr** instructions that place values to enable counting into the performance monitor and other Special Purpose Registers except for MMCR0.
4. Execute all **mtspr** instructions to initialize the performance monitor counters to the appropriate values.
5. Execute the **mtspr** instructions that place the value to enable counting into MMCR0.
6. Execute a synchronizing instruction.
7. Exit supervisor mode.
8. Start the program for which counting is to be done.

When the program being counted completes, the following steps are used to disable performance monitor counting:

1. Enter supervisor mode.
2. Execute a synchronizing instruction.
3. Execute the **mtspr** instructions that places the value to disable counting into MMCR0.
4. Execute all **mfspr** instructions to read the values from the performance monitor counters.
5. Execute a synchronizing instruction.
6. Exit supervisor mode.

The performance monitor counters contain either the number of times the selected event has occurred or the number of cycles during which the monitored event occurred after the performance monitor was enabled for counting.

Note: In either case, any counted events that occur after the performance monitor counting is enabled in supervisor mode, but before the program under study is entered, will be included in the overall count value. In the same way, any counter events that occur after the program under study is exited, but before the performance monitor counting is disabled, will also be included in the overall count value.

10.8 Performance Monitor Exceptions

The three trigger events described in *Section 10.7.2* beginning on page 334 can cause a performance monitor exception to occur and the subsequent performance monitor exception to be generated if the following sequence of actions occurs:

1. The trigger event occurs.
2. The trigger event is enabled.
3. The performance monitor exception is enabled.
4. External interrupts are enabled.

Note: This is the highest priority interrupt.

A performance monitor exception can be disabled for a given trigger event by disabling that trigger event (MMCR0[TBEE, PMC1CE, PMCjCE]). Performance monitor exceptions can be disabled for all of the trigger events collectively by disabling the performance monitor exception (MMCR0[PMXE]). The performance monitor exception, which is classified as an external interrupt, can be disabled either by disabling the performance monitor exception (MMCR0[PMXE]) or by disabling the external interrupts (MSR[EE]).

When an enabled condition or event occurs and a performance monitor exception is taken, the performance monitor exception is disabled by the hardware so that the SIAR and SDAR will contain the address and data information for an instruction that was executing at or around the time of the exception. Because the contents of the SIAR and SDAR can be altered if and only if MMCR0[PMXE] equals '1', the contents of those registers can change only if software re-enables the performance monitor exception. If such a re-enable is done and multiple performance monitor exceptions occur before the performance monitor exception is taken, then the exception reflects the most recently occurring such exception. Data from the previous exceptions are lost.

If a performance monitor exception is pending and the value of MSR[EE] is changed from '0' to '1', then the performance monitor exception will occur before the next instruction is executed provided no higher priority exception exists. The occurrence of the performance monitor exception cancels the performance monitor exception.

In summary, the following registers are set when a performance monitor exception occurs:

- SRR0[0:63] is set to the effective address of the instruction that the processor would have attempted to execute next if no interrupt conditions were present.
- SRR1[33] is set to '1' if the contents of the SDAR and the SIAR are associated with the same instruction.
- Other SRR0 and SRR1 bits are set as described in *Chapter 4 Exceptions*.
- SIAR is set to the effective address of the marked instruction, where the marked instruction is an instruction that was executing, possibly out-of-order, at or around the time that the performance monitor exception occurred. The contents of the SIAR may be altered by the processor if and only if MMCR0[PMEE] is set to '1'. Thus, after a performance monitor exception occurs, the contents of the SIAR is not altered by the processor until software sets MMCR0[PMEE] to '1'. After software sets MMCR0[PMEE] to '1', the contents of SIAR is undefined until the next performance monitor exception occurs.

- SDAR is set to the effective address of the storage operand of an instruction that was executing, possibly out-of-order, at or around the time that the performance monitor exception occurred. This storage operand is called the marked data and may be, but need not be, the storage operand (if any) of the marked instruction. If the performance monitor causes a performance monitor exception, the SRR1 indicates whether the marked data is in fact the storage operand of the marked instruction. The contents of the SDAR may be altered by the processor if and only if MMCR0[PMEE] is set to '1'. Thus, after a performance monitor exception occurs, the contents of the SDAR is not altered by the processor until software sets MMCR0[PMEE] to '1'. After software sets MMCR0[PMEE] to '1', the contents of SDAR are undefined until the next performance monitor exception occurs.
- MSR is set the same as for other external interrupts.

10.9 Instruction Matching and Sampling

The 970MP performance monitor provides a facility for the detailed analysis of instruction flow by sampling particular instructions or classes of instructions. Instructions must pass through three stages of eligibility to be marked for sampling. The contents of the SIAR/SDAR reflect the marked instruction that is currently executing.

10.9.1 Stage 1 Eligibility

There are two instruction marking facilities for stage 1:

- **IFU** - Uses the IMC array to either set or clear the mark (imr) bit associated with any matching instruction. This imr bit, along with the branch (B), first (F), split (S), and last (L) predecode bits, are retained in the L1 instruction cache along with each instruction. All instructions with the imr bit set are eligible for stage 2 of marking.
- **IDU** - Uses the BFSL predecode bits (set independent of any instruction matching in the IMC array) along with the imr_match and imr_mask fields in the MMCRA. All instructions with predecode bits (BFSL) matching imr_match when ANDed with imr_mask are eligible for stage 2 of marking.

Which facility is used depends on the MMCRA imr_select field. If imr_select equals '0', the IDU facility will be used. Otherwise, if imr_select equals '1', the IFU facility will be used.

10.9.2 Stage 2 Eligibility

Any eligible instructions from stage 1 are further filtered by the imr_mark field in the MMCRA:

'00': All IOPs
'01': Only IOPs resulting from microcode expansion
'10': Only one IOP per PowerPC instruction
'11': First IOP to go to the LSU for every PowerPC load/store instruction

10.9.3 Stage 3 Eligibility

Any eligible instructions from stage 2 are marked if the internal *ok_to_sample* performance monitor signal is asserted. This results in at most one marked instruction in the pipeline at a time. Another eligible instruction will be marked after a marked instruction completes or the previous marking cycle has timed out (set by the SCOM x'240' IDLE field).

10.10 IFU Instruction Matching Facility

The PowerPC instruction matching by opcode or extended opcode is performed by the IFU Instruction Matching facility implemented in hardware through the use of an instruction match CAM (IMC) array. When a PowerPC instruction is fetched from memory, the IFU instruction matching facility compares the instruction with the opcode/extended opcode mask values in each of the IMC array rows. If a PowerPC instruction matches one or more IMC array row masks, the IFU predecode bits associated with the marked instruction are set based on the value of the IMC function bit in each of the matched IMC array rows. The IMC function bits and descriptions of the subsequent processing for the PowerPC instruction matched for each function bit are as follows:

- Force only (fo) forces the IDU to place the PowerPC instruction in a group by itself by setting the predecode bits for the instruction as 'first' and 'last' in the group. The IMC fo function bit is used for hardware debug and workaround. It is only accessible by using scan.
- Instruction marking (imr), which is used for performance monitor instruction marking, causes the IDU to recognize that the instruction is IFU-eligible for marking. The IMC imr function bit is used by the performance monitor for marked instruction events and threshold event counts. It is accessible to the user by using the supervisor mode **mtimc**/**mfimc** instruction.

In addition to the IFU predecode bits associated with the IMC function bits, other IFU predecode bits—based on the instruction type—are bundled with each instruction in the IFU instruction cache.

Note: As long as an instruction resides in the level 1 instruction cache, its match bit will remain unchanged. If the match condition for an instruction changes, then the Level 1 instruction cache should be flushed to ensure proper setting of the match bits for all instructions.

10.10.1 Overview of the IFU Instruction Matching Facility

Each processor core includes an IFU instruction matching facility, implemented as the IMC array, which is used to maintain the kinds of IFU instruction matching requests and the mask values used for each IFU matching request. The method of reserving an IMC array row differs depending on whether the row is being requested for use by the hardware patch/debug facility (using scan only) or for the performance monitor marked instruction or threshold event facility (using the supervisor **mtimc** and supervisor or user **mfimc** instructions). All IMC array rows required for hardware patch/debug operations (fo reservations) are reserved during the 970MP power-on reset (POR) scan sequence. Any IMC array rows that have not been fo-reserved during system initial program load (IPL) may be requested by an executing program for use by the performance monitor.

Before an executing program requests an IMC array row, the program must determine which IMC array rows are available to software; that is, which rows are not fo-reserved. This information is available by reading the IMC Special Purpose Register (SPR) with the **mfimc** instruction, which may be executed only in supervisor mode. An executing program may request any IMC array rows that are not fo-reserved by writing the IMC SPR with the **mtimc** instruction, which can be executed only in supervisor mode.

10.10.2 IMC Array

The IMC array, which is contained in the IFU, consists of eight row entries as follows:

- Six rows (0 through 5) support a 17-bit partial instruction match of the opcode field in instruction bits (0:5) and extended opcode fields in instruction bits (21:31).
- Two rows (6 and 7) combined support a full 32-bit instruction match of all the instruction fields in instruction bits (0:31).

Each IMC array row includes fields for:

- The imr function bit (bit 60, called the Mark bit) that determines the predecode tag value sent to the IDU. The imr bit is programmable through the Instruction Match facility.
- The two mask values that together encode the instruction match criteria (called v0 and v1).
- The machine configuration this match applies to (PR, FP Available, VPU Available). Note that the PR bit in the MSR determines the Privileged state if a '0' and the Problem state if a '1'.
- Optional replacement field for the BFSL (predecode) bits to replace for a matched instruction (SPR cannot access theses fields).

The programmer's model view of the IMC array is shown in *Table 10-9* and *Table 10-10*.

Table 10-9. Partial Match Rows in the IMC Array

0:16	17	18	19	20	21:37	38	39	40	41	42:52	53:58	59	60	61:63
v0(17)	N/A	MSR [PR]	MSR [FP]	MSR [VPU]	v1(17)	N/A	MSR [PR]	MSR [FP]	MSR [VPU]	N/A	BFSL Replacement	Replace	Mark	Index
														0
														1
														2
														3
														4
														5

Table 10-10. Complete Match Rows in the IMC Array

0:31	32	33	34	35	36:52	53:58	59	60	61:63
v0 (Row 6) / v1 (Row 7)	N/A	MSR [PR]	MSR [FP]	MSR [VPU]	N/A	BFSL Replacement	Replace	Mark	Index
									6
									7

The following are some rules and general facts about the IMC array features and use:

- The IMC array row with full 32-bit v0 and v1 masks can be used to match the opcode, the extended opcode, and all other fields for any PowerPC instruction.
- The IMC array rows with 17-bit mask values cannot be used to match branch instructions.
- The 17-bit v0 and v1 mask values can be used to match only those bits of a PowerPC instruction (but not branch instructions) that represent the opcode and extended opcode bit fields of that instruction and cannot be used to match any other instruction bit fields such as the register fields, shift fields, mask fields, reserved fields, immediate fields, or the rc bit.
- The bits of v0 and v1 that correspond to any fields other than the opcode and extended opcode bit fields of instruction bits (21:31) should be set to the 'don't care' v0 and v1 value as is explained in *Section 10.10.5 The v0 and v1 Mask Criteria* on page 343.
- The 17-bit v0 and v1 bits (0:5) and v0 and v1 bits (6:16) correspond to instruction bits (0:5) and instruction bits (21:31).
- The 3 mode bits (PR, FP, VP) correspond to MSR[PR], MSR[FP], and MSR[VP].

10.10.3 Reading the IMC SPR with the mfimc Instruction

The IMC SPR is read in order to determine which IMC array rows are fo-reserved. The image of the IMC SPR that is obtained by executing the **mfimc** instruction (which can be executed in supervisor and user mode) is referred to as the **patch map**. The programmer's model view of the patch map is shown in *Figure 10-4*.

Figure 10-4. Patch Map

Patch Map Bit Number (IMC Array Row Address)								
0:55	56 (0)	57 (1)	58 (2)	59 (3)	60 (4)	61 (5)	62 (6)	63 (N/A)
Reserved[1]								Reserved

1. The designation 'reserved' is used both to indicate bits in the patch map that are not used for this implementation, as well as to identify the fields that are not accessible when using the **mfimc** instruction.

A patch map status bit value of '1' indicates that the associated IMC array row is fo-reserved and cannot be altered by the executing program. If the status bit value is '1', an **mtimc** to the associated IMC array row is treated as a no-op.

The facts summarized below emphasize what information is and is not available from the patch map:

- The only information that the patch map provides about the IMC array rows is whether a row is fo-reserved.
- The patch map provides no information about reservations made by the performance monitor IMR facility.
- An IMC array row that is reserved by the performance monitor IMR facility will not show a patch map status bit of '1' if the patch map is read with the **mfimc** instruction.
- There is no **mfimc**-like instruction that will help an executing program determine what v0 and v1 values were used when a performance monitor IMR facility request was made.
- All information about IMC array use must be maintained by the executing program.
- If an IMR reservation is made for an available IMC array row and the v0, v1 mask used for the IMR reservation is the same as a v0,v1 mask that is already being used for an fo reservation, the instruction selected by the v0, v1 mask will be correctly processed for both the imr request and the fo request.

As an example of the value returned by the **mfimc** instruction under a particular IMC array use scenario, assume that the following IMC array row is reserved:

- The IMC array row at address 5 is fo-reserved,

A **mfimc** instruction executed for that IMC array would return the patch map shown in *Table 10-11*.

Note: The bits for IMC array rows 1 and 4 are not set in the patch map.

Table 10-11. IMC SPR Patch Map Sample Results

Patch Map Bit Number (IMC Array Row Address)										
0		55	56 (0)	57 (1)	58 (2)	59 (3)	60 (4)	61 (5)	62 (6)	63(N/A)
Reserved			1	0	0	0	0	1	1	Reserved

10.10.4 Writing the IMC SPR With the mtimc Instruction

After an executing program determines which IMC array rows are fo-reserved (by doing a **mfimc** and seeing which patch map bits are set to '1'), the program should initialize an IMC_reservation data structure, which it should then use to track all fo/imr-reservations.

The program can make an IMR reservation of any available IMC array row through use of the **mtimc** instruction (which can be executed only in supervisor mode). The **mtimc** instruction is used to select the IMC array row, provide the v0 and v1 mask values, and set the imr bit. After a performance monitor IMR request is successfully completed, the requesting program should update its IMC_reservation data structure to record the reservation and the v0, v1 mask values.

A performance monitor IMR request remains in effect on a processor until it is:

- Canceled by an **mtimc** instruction that sets the imr bit to '0',
- Changed by an **mtimc** instruction that changes the IMC array row fields v0, v1 and writes the imr bit to '1',
- Cleared or replaced by a system reboot, or
- Overwritten by the service processor using scan or SCOM.

When an existing performance monitor IMR request is changed or canceled by a subsequent **mtimc** instruction, the executing program must update its IMC reservation data structure and invalidate the I-cache in order to reset the match bits set by a previous IMR reservation. Otherwise, stale instruction marks from the previous IMC setup might make the performance measurements unreliable, meaning old marks might still be encountered and new marks might not always occur depending on the state of the I-cache.

The programmer's model of the IMC SPR for the **mtimc** instruction differs slightly for requests of the 32-bit and the 17-bit instruction match IMC array rows. The IMC array rows for 17-bit matches, which are at IMC array row addresses 0 - 5, are written with a single **mtimc** instruction. It specifies the IMC array row address, the 17-bit opcode and extended opcode instruction mask values for v0 and v1, the machine mode mask values for v0 and v1, and it sets the imr bit to '1'. The image of the IMC SPR that is used when executing the **mtimc** instruction for IMC array row addresses 0 - 5 is shown in *Table 10-12* on page 343.

Table 10-12. IMC SPR for a 17-Bit Match

IMC Row Bit Numbers (IMC Array Row Fields)								
0:16	17	18:20	21:37	38	39:41	42:59	60	61:63
$v0_{[0\text{-}5,21\text{-}31]}$	Reserved	$V0_{PR,FP,VP}$	$v1_{[0\text{-}5,21\text{-}31]}$	Reserved	$v1_{PR,FP,VP}$	Reserved1	IMR1	IMC Row Address

1. The designation 'reserved' is used both to indicate bits in the IMC SPR that are not used for this implementation as well as to identify the fields that are not accessible when using the **mtimc** instruction.

The IMC array row for 32-bit matches, which is at IMC array row address 6 and 7, is written with two **mtimc** instructions. Each specifies an IMC array row address, a 32-bit instruction mask value, and an imr bit value as follows:

- The first **mtimc** instruction sets the IMC SPR bits (61:63) = $6_{10} = 110_2$, the IMC SPR bits (0:31) = $v0_{instruction[0\text{-}31]}$, the IMC SPR bits (33:35) = $v0_{PR,FP,VP}$, and the IMC SPR bit (60) = '0'.
- The second **mtimc** instruction sets the IMC SPR bits (61:63) = $7_{10} = 111_2$, the IMC SPR bits (0:31) = $v1_{instruction[0\text{-}31]}$, the IMC SPR bits (33:35) = $v1_{PR,FP,VP}$, and the IMC SPR bit (58) = '1'.

The image of the IMC SPR that is used when executing the first **mtimc** instruction (for IMC array row address 6) is shown in *Table 10-11* on page 342 and the image of the IMC SPR that is used when executing the second **mtimc** instruction (for IMC array row address 7) is shown in *Table 10-13*.

*Table 10-13. IMC SPR Used when Writing the Second **mtimc** Instruction for a 32-Bit Match*

IMC Row Bit Numbers (IMC Array Row Fields)							
0:31	32	33:35	36:59	60 imr	61:63 IMC Row Address		
$v1_{[0\text{-}31]}$	Reserved	$v1_{PR,FP,VP}$	Reserved	1	1	1	1

10.10.5 The v0 and v1 Mask Criteria

The mask criteria used for matching the values in the instruction bit fields and the machine state are based on the values in the v0 and v1 fields. Each pair of bits, v0(n) and v1(n); where n is 0–16, or n is 0–31, or n equals PR, FP; is interpreted as an encoded 4-value criterion. It determines how the corresponding instruction bit (m) is to be matched, where either m is 0–31 for a full instruction match or m is 0–5, 21–31 for an opcode/extended opcode match. Bit correspondences between v0, v1 bit numbers and instruction numbers depend on whether v0, v1 is a 17-bit or a 32-bit mask and are as follows:

- 17-bit mask opcode bits:
  v0, v1(0:5) corresponds to instruction bits[0:5]
- 17-bit mask extended opcode bits:
  v0, v1(6:16) corresponds to instruction bits[21:31]
- 32-bit mask full instruction bits:
  v0, v1(0:31) corresponds to instruction bits[0:31]

The bit match criteria established by the four values of v0(n), v1(n) are shown in *Table 10-14* on page 344.

Table 10-14. Encoding Bits v0 and v1 of the IMC Array Mask

v0 Value	v1 Value	Meaning
0	0	Never match (disable all)
0	1	Match a one (1)
1	0	Match a zero (0)
1	1	Always match (don't care)

10.10.6 Instruction Matching Examples

```
17-bit match using only instruction opcode (bits 0:5)
Load Doubleword: opcode = 58, extended opcode = N/A (don't care); PR=0; FP=1, VP=0/1
v0 = 0b00010111111111111 1101
v1 = 0b11101011111111111 1011

17 bit match using instruction opcode and extended opcode
Load Word and Zero Indexed: opcode = 31, extended opcode = 23; PR=1; FP=0/1, VP=0/1
v0 = 0b10000011111010001 1011
v1 = 0b01111100000101110 1111
```

10.11 IDU Instruction Sampling Facility

Another level of sampling activity—performed in the IDU—includes further processing for the imr and fo reservations made using the IFU Instruction Matching facility. Because of the way an instruction is processed through the two IDU selection stages, it is possible that the IDU instruction sampling processing can override previous IFU IMR marking.

The IDU instruction sampling facility that produces marked instructions for the instruction pipeline consists of two independent selection stages. In each of the two selection stages, the selection criteria for that stage determines which instructions pass out of that IDU selection stage as eligible to be marked. Because an instruction passes through each of the two IDU selection stages after it is processed for IFU IMR marking, it is possible that the IDU instruction sampling eligibility criteria can override previous IFU IMR marking. It is also possible that the IDU stage 2 processing can override IDU stage 1 processing if the criteria for each of the two selection stages are not set up correctly.

An eligible instruction that is marked by the IDU is referred to as a marked (sampled) instruction.

10.11.1 Overview of the IDU Instruction Sampling Facility

The IDU instruction sampling facility uses the IFU imr and predecode bits from the instruction cache, together with PMU/SCOM data and control fields, to determine which instructions are eligible to be marked (sampled) and when an instruction can actually be marked (sampled).

Operation of the IDU instruction sampling facility to determine which instructions are eligible for marking occurs continuously when performance monitor mode sampling is enabled (MMCRA[63] equals '1'). The choice of which instructions are eligible to be marked is based on the values of the IFU imr and predecode bits combined with the values of the select, mask, match, mark, and filter fields. The IDU continuously desig-

nates instructions as eligible to be marked based on the above fields, but the IDU only marks instructions when sampling is enabled and the performance monitor signals the IDU that it is *ok_to_sample*. Only one marked group flows through the instruction pipeline at a time.

IDU processing of an instruction based on fo IMC function bit is independent of IDU processing based on the IFU imr function bit, so a given instruction might be processed by the IDU for any or all of the fo and imr functions.

The IDU eligibility stages continuously produce eligible instructions. The sample_enable field combined with the performance monitor signal *ok_to_sample* control final marking as outlined below. The information in parenthesis corresponds to the annotations on the left in *Figure 10-5* on page 349.

- The imr_select field (MMCRA[49]) determines the method used for stage 1 instruction eligibility (eligibility stage 1 - method).
- Depending on the imr_select field value, the imr_mask field (MMCRA[52:55]) and the imr_match field (MMCRA[56:59]) can be used to determine the type of the stage 1 eligible instructions that pass through to stage 2 (eligibility stage 1 - type).
- The imr_mark field (MMCRA[50:51]) determines what type of stage 1 eligible instructions are to be considered for stage 2 eligibility (eligibility stage 2 - type).
- The imr_filter field (SCOM x'34' [11:12]) determines which and how many of the stage 2 eligible instructions actually become marked instructions in the pipeline (eligibility stage 2 - method).
- The performance monitor signal *ok_to_sample* determines whether marking is blocked or might resume (mark/no mark stage).

The IFU matching and the IDU instruction sampling activities occur continuously regardless of whether instructions are being marked by the IDU. The IDU can mark an instruction only when the performance monitor signals that marking is enabled. After an instruction is marked by the IDU, the performance monitor disables marking until either a marked instruction completes, the instruction is a store that is sent to the STS, or the performance monitor completion delay timer indicates that the marked instruction has been flushed.

Note: As long as an instruction resides in the Level 1 instruction cache, its match bit will remain unchanged. If the match condition for an instruction changes, the Level 1 instruction cache should be flushed to ensure proper setting of the match bits for all instructions.

10.11.2 Stage 1 Eligibility

The IDU uses the IFU predecode bits for branch, first, split, and last (shown in *Table 10-15* on page 346) and the imr bit stored with an instruction in the instruction cache to establish eligibility for marking.

Note: As long as an instruction resides in the Level 1 instruction cache, its imr match bit will remain unchanged. If the match condition for an instruction changes, the Level 1 instruction cache should be flushed to ensure proper setting of the imr match bits for all instructions.

The method used to choose IDU stage 1 eligible instructions is based on the value of the imr_select field (MMCRA[49]). Depending on the value of the imr_select field, a second decision point may be required to choose the type of stage 1 eligible instructions using the imr_mask field (MMCRA[52:55]) and the imr_match field (MMCRA[56:59]). These two scenarios, based on the value of the imr_select field value, are as follows:

- imr_select equals '1':
  The IFU imr bit is used to determine stage 1 eligibility. All instructions with the IFU imr bit set are passed through to stage 2 as eligible for marking.

- imr_select equals '0':
  The IFU BFSL predecode bits are used to determine stage 1 eligibility.

If imr_select equals '1', so that the IFU imr bit determines stage 1 eligibility, it is possible to choose values for imr_mark (IDU eligibility stage 2 - type) that will cancel the IMR eligibility created in stage 1.

If imr_select equals '0' and the IFU BFSL predecode bits are used to determine stage 1 eligibility, there are two further stages of processing to establish the type of instructions that are eligible:

1. The BFSL predecode bits for the instruction are ANDed with the imr_mask field to produce a 4-bit intermediate result, and
2. The 4-bit intermediate result is compared with the imr_match field. All instructions with an exact 4-bit match between the intermediate result and the imr_match field are passed through to stage 2 as eligible for marking.

To match all instructions, and thus pass all instructions through to stage 2 as eligible for marking, set the following values for the stage 1 method/type decision variables:

- imr_select: '0'
- imr_mask: '0000'
- imr_match: '0000'

The IFU BFSL predecode will be used, the mask will result in all zeros for the intermediate result, and the match will always succeed. The eligibility method chosen at stage 1 can determine what kind of instruction is counted for the performance monitor count event called "number of instructions completed," depending on how the eligibility criteria is set up in stage 2.

Table 10-15. IFU BSFL Predecode Bit Definitions (Page 1 of 2)

B(ranch)	S(plit)	F(irst)	L(ast)	Classification	Description
0	0	0	0	Simple	One IOP formed from one instruction with restrictions.
0	0	0	1	Simple-Last	IOP formed will be the last in the resultant dispatch group.
0	0	1	0	Simple-First	IOP formed will be the first in the resultant dispatch group.
0	0	1	1	Simple-Only	IOP formed will be the only in the resultant dispatch group.
0	1	0	0	Split	Two IOPs formed from one instruction; both must be in the same dispatch group.
0	1	0	1	Split-Last	Two IOPs formed from one instruction; the second IOP must be the last IOP in the resultant group.
0	1	1	0	Split-First	Two IOPs formed from one instruction; the first IOP must be the first IOP in the resultant dispatch group.
0	1	1	1	Split-Only	Two IOPs formed from one instruction; no other IOPs are present in the resultant dispatch group.
1	0	0	0	Branch - Conditional	IOP formed from a conditional branch instruction.
1	0	0	1	Branch - Unconditional	IOP formed from an unconditional branch instruction.
1	0	1	0	Illegal Opcode	Not a valid instruction.
1	0	1	1	Reserved	
1	1	0	0	Microcode - Hard	A nonprogrammable microcode sequence must be generated.

1. Field mask used to identify the Condition Register (CR) fields to be updated by the **mtcrf** instructions.

Table 10-15. IFU BSFL Predecode Bit Definitions (Page 2 of 2)

B(ranch)	S(plit)	F(irst)	L(ast)	Classification	Description
1	1	0	1	Microcode - Soft	A system software programmable microcode sequence must be generated.
1	1	1	0	Microcode - Conditional (otherwise: Split-Last)	A nonprogrammable microcode sequence must be generated if certain conditions are not met.
1	1	1	1	Microcode - Conditional (otherwise: Split-Only)	A non-programmable microcode sequence must be generated if certain conditions are not met (for example, the FXM[1] field is not singular).

1. Field mask used to identify the Condition Register (CR) fields to be updated by the **mtcrf** instructions.

10.11.3 Stage 2 Eligibility

The imr_mark field (MMCRA[50:51]) value and then the imr_filter field (SCOM x'340'[11:12]) are used by the IDU to establish stage 2 eligibility for marking. The first decision point for IDU stage 2 eligibility is the type of stage 1 eligible instructions that can be stage 2 eligible; this determines final stage 2 eligibility. The second decision point for IDU stage 2 eligibility is the method of passing the eligible information to the mark/no mark stage; this determines what eligible instructions actually get marked.

The type of stage 1 eligible instructions that will be stage 2 eligible is based on the value of the imr_mark field (MMCRA[50:51]). The imr_mark field value determines stage 2 eligibility of instructions as follows:

00 All stage 1 eligible IOPs are stage 2 eligible for marking.

01 Only stage 1 eligible IOPs that resulted from microcode expansion are stage 2 eligible for marking.

10 Only one IOP per stage 1 eligible PowerPC instruction is stage 2 eligible for marking (use this mode to make all PowerPC instructions eligible).

11 For every stage 1 eligible PowerPC instruction, the first IOP that goes to the LSU is stage 2 eligible for marking (use this mode to make all load/store instructions eligible).

The stage 1 eligibility method combined with the stage 2 eligibility type determines what kind of instruction is counted for the performance monitor count event called "number of instructions completed" as shown in *Section 10.11.6 Examples of Instruction Sampling Scenarios* on page 351.

After the type of stage 2 eligible instructions is established, the method of passing the stage 2 eligibility information to the mark/no mark stage is determined in two steps using the imf_filter[11] bit value and then using the imr_filter[12] bit value, which have the following functions:

imr_filter[11:12]

0x No filtering

1x Randomly sample eligible IOPs

Bit 12 selects one of two behaviors in sampling from microcode expansions (and has no effect on sampling from non-microcode groups):

10	Use the Good_Address mode of sampling microcode expansions
11	Use the More_Hits mode of sampling microcode expansions

In Good_Address mode, there is at most one IOP in any microcode expansion that is eligible for sampling. This is either the first load/store IOP if there are any load/store IOPs in the expansion, or the first IOP in the final group of the expansion. If the random filter suppresses marking this IOP, then no IOP will be marked for the microcode expansion.

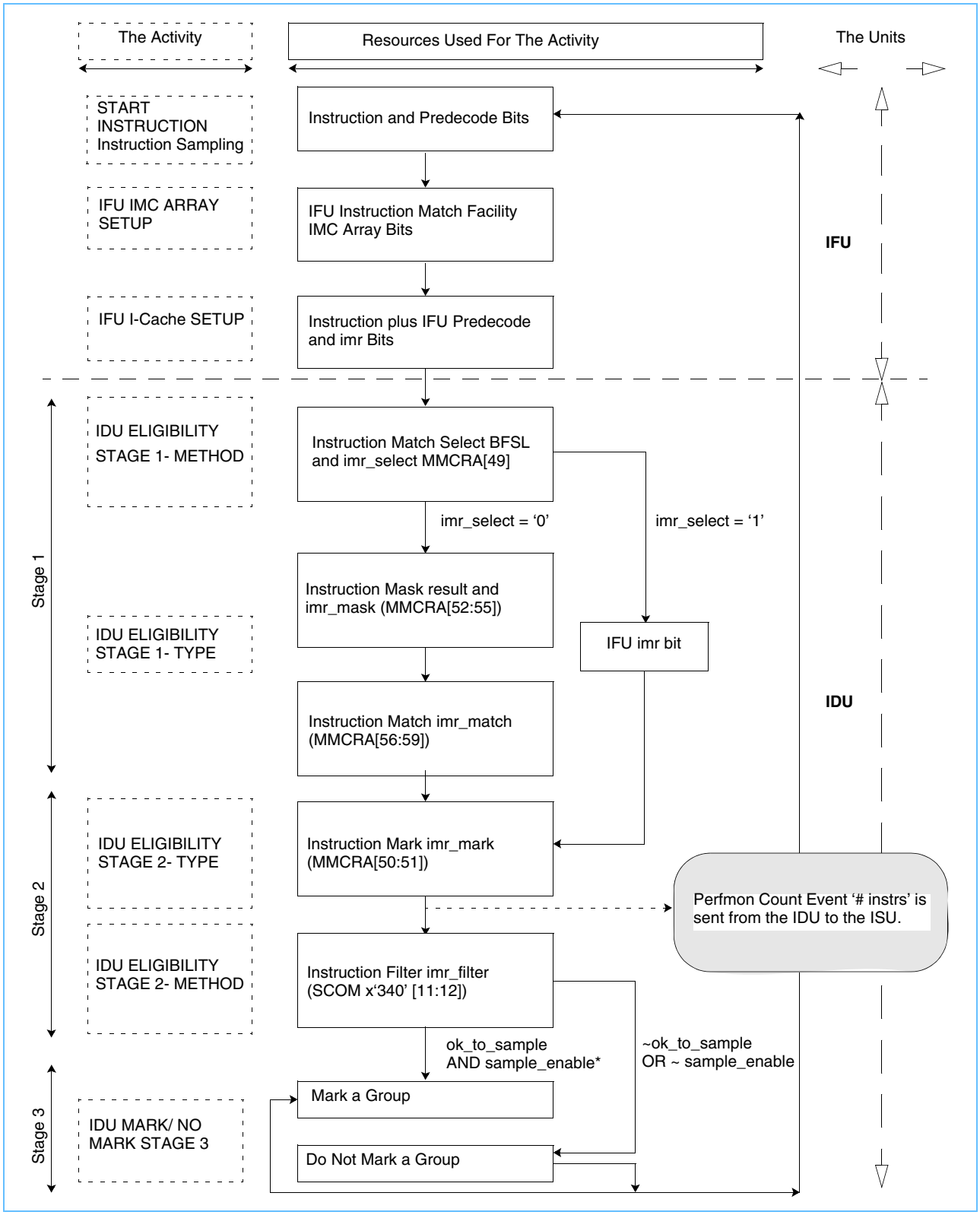
In More_Hits mode, multiple IOPs in a microcode expansion are eligible for sampling: the first load/store IOP in any group, or the first IOP of the final group. If the random filter suppresses marking the first of these IOPs, a subsequent one might still be sampled. (However, at most one will be marked in a single microcode expansion.)

The suggested mode for imr_filter[11:12] is '10'.

10.11.4 Stage 3 Mark/No Mark

The sample_enable field (MMCRA[63]) value and the *ok_to_sample* signal state are used by the IDU to complete the mark/no mark stage for instruction sampling. If marking is disabled with the sample_enable bit (MMCRA[63] equals '0'), then no group is marked by the IDU regardless of stage 2 eligibility. If marking is enabled with the sample_enable bit (MMCRA[63] equals '1'), then marking depends on the state of the handshake protocol between the IDU and the performance monitor signal *ok_to_sample*. The *ok_to_sample* signal is sent by the performance monitor to the IDU when the performance monitor determines that the previous marking cycle has completed successfully or has timed out. The handshake and synchronization mechanism are explained in *Section 10.11.5 Complete Masking, Matching, and Marking Cycle* on page 350.

Figure 10-5. IFU and IDU Instruction Sampling Flow



10.11.5 Complete Masking, Matching, and Marking Cycle

Instruction marking is set up by initializing the MMCRx[imr_xxx] fields, the SCOMyy[imr_filter] fields, and possibly the SCOMxx[xx_delay] for the kind of marking to be done. *Section 10.11.7 Enabling and Disabling Marking* on page 354 describes the procedure for initializing these registers plus any other registers to support performance monitor counted events. There are several examples at the end of this section that show the values of the imr_xxx fields for common marking and counting scenarios.

Once the instruction marking cycle is set up and enabled, instructions are continuously processed for eligibility as is described in *Section 10.11.2 Stage 1 Eligibility* on page 345, and in *Section 10.11.3 Stage 2 Eligibility* on page 347. The actual marking of a group described in *Section 10.11.4 Stage 3 Mark/No Mark* on page 348. The steps that occur from when the performance monitor *ok_to_sample* signal initiates marking until the next *ok_to_sample* signal initiates the next marking cycle are described in this section.

The overall timing of the marking cycle is driven by two performance monitor timers called the completion_delay counter and the idle_delay counter. The completion_delay counter is first initialized at the start of a marking cycle from the value in the sampling logic completion delay field (SCOM x‘240’ [COMPLN]). The idle_delay counter is initialized at the end of a marking cycle from the value in DELAY field (SCOM x‘240’[DELAY]).

The purpose of the completion_delay counter is to predict the situation that the marked instruction has been flushed from the instruction pipeline before it can complete. The purpose of the idle_delay counter is to introduce a period of time between the end of a marking cycle (through either instruction completion or flush) and the start of the next marking cycle. The completion_delay and the idle_delay values must be greater than zero whenever matching or marking is enabled. Otherwise, the processor activity will be undefined.

A marking cycle begins when the performance monitor asserts the *ok_to_sample* signal for one cycle. The assertion of this signal (assuming that sample_enable equals ‘1’) causes the IDU to mark the next group that enters stage 3 if it has passed all the stage 2 eligibility tests. After a group is marked in the IDU (this is a performance monitor count event), the IDU continues to process instructions for eligibility but does not mark another group until the next *ok_to_sample* signal is received from the PMU.

When the signal indicating that the group is marked in the IDU is sent from the IDU to the performance monitor (this is also a performance monitor count event), the performance monitor begins decrementing the completion_delay counter by one each cycle that the signal group_completed is asserted. If group_completed is not asserted, the completion_delay counter is not decremented.

This use of the completion_delay counter is intended to model a marked_group_flushed situation. The underlying assumption of this model and the default value of COMPLN equals 20 is as follows: if no marked group event occurs in any functional unit during a full wrap of the 20-entry completion buffer, then the marked group has been flushed. The completion_delay value must be greater than zero whenever instruction sampling is enabled or processor activity will be undefined.

If the completion_delay counter does not time out between when the signal indicating that the group is marked in the IDU is received by the performance monitor and when the next marked event signal is received by the performance monitor, the completion_delay timer is reset to the value in COMPLN. It resumes decrementing for each cycle that group_completed is asserted. At each stage of the marking cycle, the completion_delay counter is initialized, counts down whenever the signal group_completed is asserted, and either times out or is re-initialized when the next marked event occurs. Thus, at each stage of the marking cycle, the marked group is allowed the COMPLN number of cycles while groups are completing from the completion buffer before the marked group is considered flushed.

The sequence of events for a complete marking cycle, where a marked group moves through all of the instruction pipeline stages without being flushed, is as follows:

1. The performance monitor signal *ok_to_sample* is asserted for one cycle.
2. A group is marked when the IDU transfers the group to the ISU (performance monitor count event direct5[4]). The completion_delay counter is initialized to the value COMPLN and begins to decrement on each cycle that group_completed is asserted.
3. A marked group is dispatched (performance monitor count event direct1[2]). The completion_delay counter is reinitialized to the value COMPLN and begins to decrement on each cycle that group_completed is asserted.
4. A marked group is issued (performance monitor count event direct6[5]). The completion_delay counter is reinitialized to the value COMPLN and begins to decrement on each cycle that group_completed is asserted.
5. A marked group finishes (FPU: performance monitor count event direct7[4], FXU: performance monitor count event direct6[4], CRU: performance monitor count event direct4[5], BRU: performance monitor count event direct2[5], LSU: performance monitor count event direct8[4], any unit: performance monitor count event direct7[5]). The completion_delay counter is reinitialized to the value COMPLN and begins to decrement on each cycle that group_completed is asserted.
6. A marked group completes (performance monitor count event direct4[4]).
7. The idle_delay counter is initialized to the value DELAY and is then decremented to '0'.
8. The *ok_to_sample* signal is asserted for one cycle and the marking cycle begins again at step 2.

When the completion_delay counter times out (performance monitor count event direct5[5]), the performance monitor enters the marking cycle state where the idle_delay counter is set to the value DELAY and is then decremented to '0' (step 7 above). When the idle_delay counter times out, the performance monitor asserts the *ok_to_sample* signal for one cycle, and the marking cycle begins again.

The default idle_delay value is set to four. The idle_delay value must be greater than zero whenever matching/marking is enabled or processor activity will be undefined.

If a marked store is sent to the STS (performance monitor count event direct6[3]), the event called "marked store sent to STS" stops the marking cycle described above and causes the performance monitor to enter the idle state. The performance monitor stays in the idle state until one of the signals "sampled store complete" (performance monitor count event direct1[3]) or "sampled store complete with intervention" (performance monitor count event direct3[3]) is received. At that time, the performance monitor resumes the marking cycle at marking cycle step 7 above.

When sampling_enable is set to zero, the performance monitor enters the idle state of the marking cycle until sampling is enabled again.

10.11.6 Examples of Instruction Sampling Scenarios

Follow the procedure in *Section 10.11.7 Enabling and Disabling Marking* on page 354 to place the values for instruction sampling into the appropriate Special Purpose Register fields. *Section 10.4 Performance Monitor Control Registers* on page 301 describes the performance monitor related registers and fields. To count marked events, the appropriate PMCxSEL fields should also be set up and the enable counting bit must be set to the enabled value. In each of the examples below, only the values of the Instruction Sampling Register fields are shown.

Example 1: Set up the instruction sampling facility to count PowerPC instructions as the performance monitor count event called "number instructions completed."

The values given here are those that set up the instruction sampling stages so that the performance monitor count event called "number instructions completed" will be the count for PowerPC instructions completed. If sampling is enabled, instructions will be randomly sampled. This is the recommended setting. The required field values are as follows:

imr_mark	'10'	Only one IOP per PowerPC instruction is stage 2 eligible for marking.
imr_select	'0'	
imr_mask	'0000'	
imr_match	'0000'	
imr_filter[11:12]	'10'	If sampling is enabled, randomly sample.

Example 2: Set up the instruction sampling facility to count IOP instructions as the performance monitor count event called "number instructions completed."

The values given here are those that set up the instruction sampling stages so that the performance monitor count event called "number instructions completed" will be the count for IOP instructions completed. The required field values are as follows:

imr_mark	'00'	All stage 1 eligible IOPs are stage 2 eligible for marking.
imr_select	'0'	
imr_mask	'0000'	
imr_match	'0000'	

Example 3: Set up the instruction sampling facility to count load/store instructions as the performance monitor count event called "number instructions completed."

The values given here are those that set up the instruction sampling stages so that the performance monitor count event called "number instructions completed" will be the count for load/store instructions completed. The required field values are as follows:

imr_mark	'11'	For every PowerPC load/store instruction, the first IOP that goes to the LSU is stage 2 eligible for marking.
imr_select	'0'	
imr_mask	'0000'	
imr_match	'0000'	

Example 4: Set up the instruction sampling facility to match or mask all PowerPC instructions that are BFSL-Split, and then sample eligible instructions that get through the random filter.

The values given here are those that set up the instruction sampling stages so that the performance monitor count event called "number instructions completed" will be the count for PowerPC BFSL-Split instructions completed. The required field values are as follows:

imr_mark	'00'	All stage 1 eligible IOPs are stage 2 eligible for marking.
imr_select	'0'	
imr_mask	'0100'	
imr_match	'0100'	
imr_filter[11:12]	'10'	If sampling is enabled, randomly sample.

Example 5: Set up the instruction sampling facility to match or mask all PowerPC instructions that are BFSL-Hard microcoded, and then sample all eligible instructions.

The values given here are those that set up the instruction sampling stages so that the performance monitor count event called "number instructions completed" will be the count for PowerPC BSFL-Hard microcoded instructions completed. The required field values are as follows:

imr_mark	'00'	All stage 1 eligible IOPs are stage 2 eligible for marking.
imr_select	'0'	
imr_mask	'1100'	
imr_match	'1100'	
imr_filter[11:12]	'00'	Pass all stage 1 eligible bits in the group.

Example 6: Set up the instruction sampling facility to IFU IMR match or mask all PowerPC add instructions, and then sample all eligible instructions.

The values given here are those that set up the instruction sampling stages so that the performance monitor count event called "number instructions completed" will be the count for IMC-marked PowerPC instructions completed. The required field values are as follows:

imr_mark	'10'	
imr_select	'1'	
imr_mask	'0000'	
imr_match	'0000'	
imr_filter[11:12]	'00'	If sampling is enabled, randomly sample.

10.11.7 Enabling and Disabling Marking

The processor comes out of reset with instruction marking disabled (MMCRA[63] equals '0') and with all of the MMCRA/SCOM[imr_xxx] fields set to zero. To set up the performance monitor for marking, follow these steps:

1. Enter supervisor mode.
2. Execute a synchronizing instruction or wait for any previous activity to complete.
3. Execute all **mtspr** instructions that place values to set up for marking (and counting if that is to be done) into the Performance Monitor Registers *except* for the counting enable in MMCR0 and the sample_enable in MMCRA.
4. Wait for all previous **mtspr** instructions to complete, and then execute the **mtspr** instruction to enable counting in MMCR0.
5. Wait for the previous **mtspr** instruction to complete, and then execute the **mtspr** instruction that enables marking in MMCRA.
6. Execute a synchronizing instruction or wait for the last **mtspr** instruction to complete.
7. Exit supervisor mode.
8. Start the program for which marking (and counting) is to be done.

Note: Any marked or counted events that occur after the performance monitor counting is enabled in supervisor mode, but before the program under study is entered, will be included in the overall mark or count activity. In the same way, any counter events that occur after the program under study is exited, but before the performance monitor marking or counting is disabled, will also be included in the overall mark/count activity.

When the program being marked or counted completes, the following steps disable performance monitor marking or counting:

1. Enter supervisor mode.
2. Wait for the previous **mtspr** instructions to complete, and then execute the **mtspr** instruction that disables marking in MMCRA.
3. Wait for the previous **mtspr** instruction to complete, and then execute the **mtspr** instruction to disable counting in MMCR0.
4. Execute a synchronizing instruction or wait for the last **mtspr** instruction to complete.
5. Wait for the counting operations that are in flight to complete and then execute the **mfspr** instructions to read the values from the performance monitor counters.
6. Execute a synchronizing instruction or wait for the last **mfspr** instruction to complete.
7. Exit supervisor mode.

Notes:

If marking is disabled while a marked instruction is still active, the performance monitor will finish that marking operation in the typical way. The marking state machine for the performance monitor will then go to idle until marking is again enabled.

Instruction marking is disabled while in Single Step or Branch trace mode.

10.12 SIAR and SDAR Registers

The Sampled Instruction Address Register (SIAR) and the Sampled Data Address Register (SDAR) are used, respectively, to save the effective address of a sampled instruction and the effective address of a storage operand for a sampled instruction when the processor is in either trace-marking mode or instruction-sampling mode. The terms "sampled" and "marked" are use interchangeably.

The processor is in instruction-sampling mode whenever MMCRA[63] equals '1' and MSR[SE] and MSR[BE] equal '0'. The processor is in a well-defined trace-marking mode whenever MMCRA[63] equals '0' and either MSR[SE] equals '1' or MSR[BE] equals '1', or MSR[SE] and MSR[BE] equal '1'.

Note: If instruction sampling is not disabled during trace-marking by setting MMCRA[63] to '0', results are undefined.

The contents of the SIAR and SDAR depend on the marking modes that the processor is in, as explained in the following sections.

10.12.1 Instruction Sampling

When the processor is not in trace-marking mode and instruction-sampling mode is enabled, instruction-sampling mode is active regardless of whether the performance monitor is enabled for any counting activity. In instruction-sampling mode, the performance monitor interacts with the IDU to initiate the instruction-sampling cycle. It then monitors the progress of the sampled instruction as it moves through the instruction pipeline. Each instruction-sampling cycle ends either when the marked instruction completes or when the performance monitor determines that the marked instruction has been flushed from the pipeline.

10.12.1.1 Performance Monitor Exceptions

Performance monitor exceptions occur when a performance monitor counter becomes negative and the counter negative exception is enabled, or when a time-base event occurs and the time-base exception is enabled. When a performance monitor exception occurs, SIAR and SDAR have the following values:

- The SIAR contains the effective address of the last sampled instruction.
- The SDAR is set to the effective address of the storage operand of the last sampled instruction issued to the LSU.
- The effective address of the storage operand contained in the SDAR might be, but need not be, associated with the SIAR instruction as explained above.
- If single step or branch trace (SE/BE) tracing is inactive, the contents of the SIAR and the SDAR are frozen when a performance monitor exception is raised, at which time the hardware sets MMCR0[PMXE] to '0' (locking the SIAR and SDAR).
- If SE/BE tracing is active, the contents of the SIAR, the SDAR, and SRR1[33] as used by the performance monitor exception facility are undefined and can change even when performance monitor exceptions are disabled (MMCR0[PMXE] equals '0').
- If SE/BE tracing is inactive, the contents of SIAR and SDAR remain frozen until software sets MMCR0[PMXE] to '1'. The contents of SIAR and SDAR can be altered by the processor if and only if MMCR0[PMXE] equals '1' provided SE/BE tracing is inactive.
- If the performance monitor exception is enabled and MSR[EE] equals '1', the performance monitor exception condition causes a performance monitor exception to be taken and the value of SRR1[33] indicates whether the contents of the SIAR and SDAR refer to the same instruction.

- If SRR1[33] equals '1', and SE/BE tracing is inactive, and there was only one sampled instruction in the machine, the SDAR and SIAR contents are associated with the same instruction.
- If SE/BE tracing is inactive, when SRR1[33] equals '0' it indicates either that the SIAR and SDAR contents are not associated with the same instruction or that the SIAR instruction had no storage operand.
- After software sets MMCR0[PMXE] to '1' and if SE/BE tracing is inactive, the contents of SIAR and SDAR are undefined with respect to performance monitor exception processing until the next performance monitor exception occurs.
- After software sets MMCR0[PMXE] to '1' and if SE/BE tracing is inactive, the contents of SIAR and SDAR are again available for use by the performance monitor instruction-sampling facility as described above.

10.12.2 Single Step and Branch Trace Marking Mode

When the processor is in SE or BE trace mode, instruction-sampling activity on the performance monitor is disabled, but all other performance monitor activities (except sampling) can be active. In particular, performance monitor count events for marked instructions will still be processed if counting is enabled. The marked events counted when trace mode is active will be for trace-marked events, not for sampled events.

In BE mode, the performance monitor count event called "number of instructions completed" is not accurate. It is possible for software to calculate the number of instructions in a basic block by capturing the starting address of the basic block from the SRR0 value of the previous branch and using it together with the ending address of the basic block from the SIAR value.

10.12.2.1 Single Step Trace Mode

If MSR[SE] equals '1', the processor is in single step trace mode. Every instruction is trace-marked. The processor is forced into single instruction mode regardless of the value of the IFU predecode bits. An instruction is trace-marked when it is transferred from the IDU to the ISU and SRR1[33] is reset to zero. If the PowerPC instruction spans multiple groups, the first load/store IOP or the first IOP in the last group is the one marked.

The SIAR is updated at dispatch to contain the address for the trace-marked instruction. The SDAR is updated by the LSU if the PowerPC instruction includes one or more load/store operation. If the SDAR is updated by the LSU, it contains the address of the storage operand for the first load/store IOP. If the SDAR is updated by the LSU, the LSU also causes SRR1[33] to be set to '1' to indicate that the contents of the SIAR and the SDAR are associated with the same trace-marked instruction.

When the trace-marked instruction completes, the processor generates a trace exception. During trace exception processing, the SIAR value is that for the trace-marked instruction that has just successfully completed. During trace exception processing, the SDAR value—if it was updated for this instruction—is that of the first load/store operation of the trace-marked instruction that has just successfully completed. A global flush is performed after the return from trace exception processing.

In the interval between the **rfid** for the trace exception processing for the last completed instruction and the dispatch of the next instruction, the SIAR and SDAR contents represent the last instruction completed and not the instruction that is moving through the IFU to the IDU for dispatch. The next instruction that will be dispatched after the global flush is trace-marked when it is transferred from the IDU to the ISU, SRR1[33] is reset to zero ('0'), and the SIAR value is set for that next marked instruction. The single instruction trace-marking cycle continues as described above.

10.12.2.2 Branch Trace Mode

If MSR[BE] equals '1', the processor is in branch trace mode. Every branch instruction is trace-marked. A branch instruction is trace-marked when it is transferred from the IDU to the ISU.

The SIAR is updated at dispatch to contain the address for the trace-marked branch instruction. The SDAR contents are undefined. When the trace-marked branch instruction completes, the processor generates a trace exception. During trace exception processing, the SIAR value represents the trace-marked branch instruction that has just completed successfully. A global flush is performed after the return from trace exception processing.

In the interval between the **rfid** for the trace exception processing for the last completed branch instruction and the dispatch of the next branch instruction, the SIAR contents represent the last branch instruction completed. The next branch instruction decoded by the IDU after the global flush is trace-marked when it is transferred from the IDU to the ISU and the SIAR value is set for that next marked branch instruction. The branch trace-marking cycle continues as described above.

10.12.3 Comparison to Previous PowerPC Processors

According to the PowerPC Architecture, the SIAR contains the effective address of an instruction that was executing *around* the time of a performance monitor exception. The PowerPC 604 and POWER3 processors only updated the Sampled Instruction Registers for sampled instructions (although their sampling method is different). The RS64 processors accomplished this by updating the SIAR and SDAR with the address of the last instruction to complete when a performance monitor exception occurs.

On previous processors that had relatively short pipelines and few instructions in flight, the sampled instruction was at most 20 or so instructions away from the instruction that caused the exception. On the 970MP microprocessor, with the potential for over a hundred instructions in flight, that distance grows. The theoretical maximum is once every 200 - 250 instructions, while the likely distance is 50 - 75 instructions.

Performance profiling tools that use performance monitor events to determine when the SIAR and SDAR are read (for example, read SIAR every 100 L1 D-cache misses) can profile based on any performance monitor event. To assure accuracy, however, only *sampled* events should be profiled. These are a subset of all events that are caused by sampled instructions. The SIAR is set by a sampled instruction, so you can be fairly sure that when an exception caused by a sampled event (a counter overflowing for example), the SIAR is pointing to the *exact* instruction that caused it. In this case, the 970MP microprocessor is more accurate than previous processors. If you profile on non-sampled events, you cannot be sure that the exception was caused by the instruction (group actually) pointed to by the SIAR. The offending instruction was executing *around* the sampled instruction, depending on the event, probably within 50 - 100 instructions.

10.13 Thresholding

Thresholding can be used to obtain counts of the number of marked instructions for which the execution time between a designated start/end pipeline event pair exceeds a specified threshold value. Only one marked instruction is active in the 970MP processing unit pipeline at a time, and only one threshold value can be used for comparison with the selected start/end event pair. The start and end events that can be used for thresholding are shown in *Figure 10-6 Performance Monitor Threshold Logic* on page 358. The values of the respective threshold start/end bit fields in MMCRA[THRSTRT,THREND] are shown in *Table 10-16 Start and End Event Select Bits and the Performance Monitor Threshold Logic* on page 359. The threshold value is specified in the MMCR1[THRSHOLD] field. The threshold value can be further scaled by HID0[13]. If

HID0[13] equals '0', it causes the thresholder to count every processor cycle. If HID0[13] equals '1', it causes the thresholder to count every 32 processor cycles. For a marked instruction moving through the 970MP processing unit pipeline, the events that can be used for threshold start/end measurement occur in the following order: marked in IDU, dispatch, issue, finish, complete.

Once a pair of start/end threshold events is selected and the start event occurs, the threshold facility begins decrementing from the threshold value and continues to decrement until either the decrementer times out or the end event occurs. If the decrementer times out and if the threshold logic event is selected for counting, the threshold logic event counter is incremented. Both the thresholder time out and the occurrence of the end event cause the threshold decrementer to be reset to the threshold value. The thresholder begins decrementing when the next start event occurs.

Threshold start/end pairs must be selected in a manner that represents a reasonable scenario. For example, a start event that is the same as the end event will not provide useful threshold event count information regardless of the threshold value selected. A start event that occurs later in the pipeline than the end event will not give a useful measure of the transit time of a marked instruction through the pipeline. The results of unreasonable threshold start/end event selections may produce undefined results.

Figure 10-6. Performance Monitor Threshold Logic

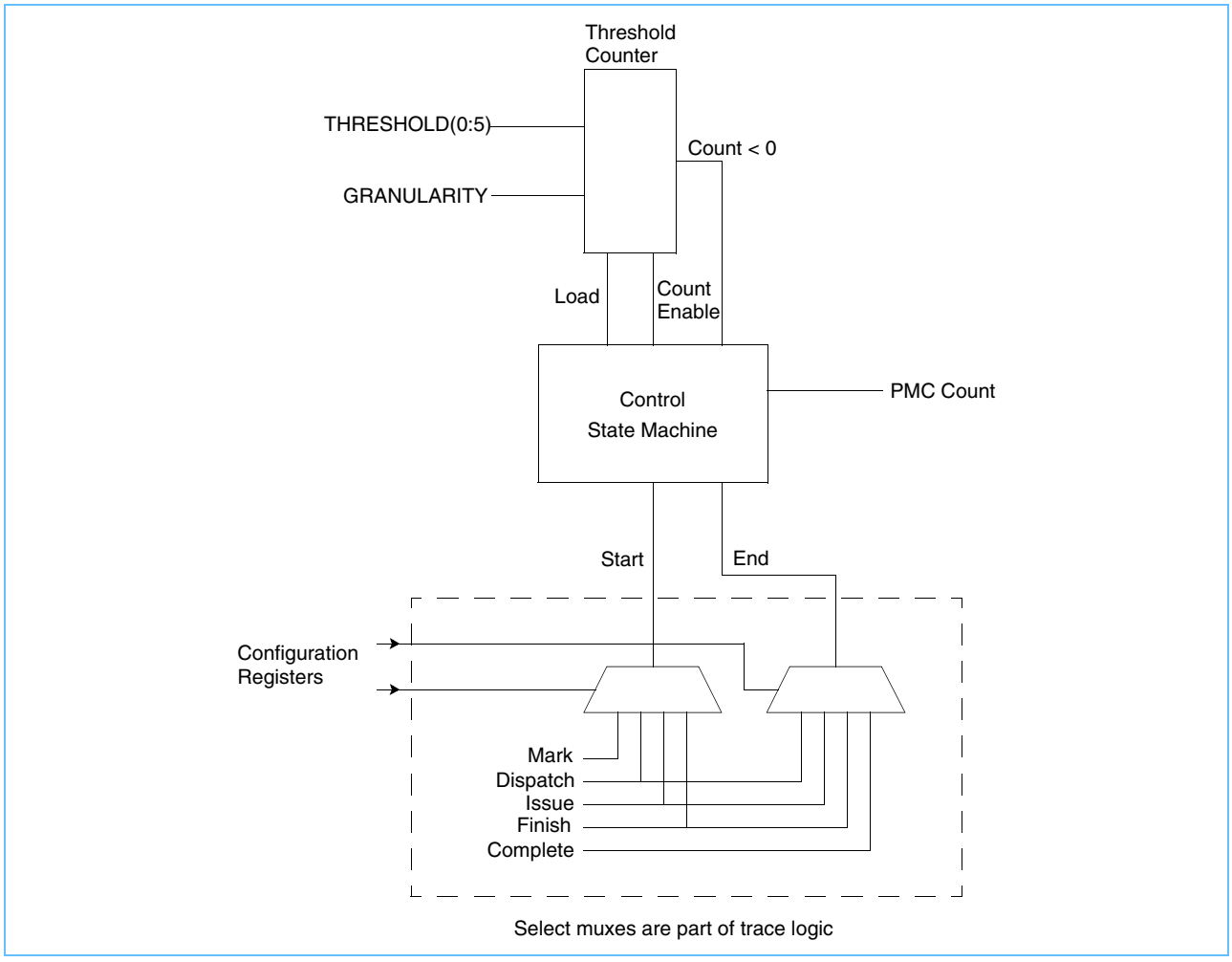


Table 10-16. Start and End Event Select Bits and the Performance Monitor Threshold Logic

MMCR1 [THRSTRT] Value	Threshold Start Event Selected	MMCR1 [THREND] Value	Threshold End Event Selected
000	No start event	000	No end event
001	Group marked in IDU	001	No end event
010	Marked group dispatched	010	Marked group dispatched
011	Marked group issued	011	Marked group issued
100	Marked group finish[1]	100	Marked group finish
101	Marked group complete[1]	101	Marked group complete
110	No start event	110	No end event
111	No start event	111	No end event

1. An instruction that has finished but not completed has gone all the way through the pipeline, and the renamed registers have been updated with new values. However, it is still sitting in the completion queue. When an instruction completes, the architected registers are updated with the values from the renamed registers.

10.14 Detailed Event Information

Table 10-17. Detailed Event Descriptions (Page 1 of 8)

Event Description	Detailed Description
CLB has x where x is 0 to 8	The cache line buffer (CLB) is a 4-instruction wide by 8-instruction deep buffer between the fetch unit and the dispatch unit. This signal indicates how many entries, each of which is 4-instructions wide, are occupied at any given time.
Valid instructions available, but IFU held by BIQ or IDU	This signal is asserted each time either the IDU is full or the branch instruction queue (BIQ) is full.
Branch execution issue valid	This signal is asserted each time the ISU issues a branch instruction.
Branch miss predict due to CR value	This signal is asserted when the branch execution unit detects a branch mispredict because the CR value is the opposite of the predicted value. This signal is asserted after a branch issue event and results in a branch redirect flush if not overridden by a flush of an older instruction.
Branch miss predict due to target address prediction	This signal is asserted each time the branch execution unit detects an incorrect target address prediction. This signal is asserted after a valid branch execution unit issue and causes a branch mispredict flush unless a flush is detected from an older instruction.
CR mapper full	The ISU sends a signal indicating that the CR mapper cannot accept any more groups. Dispatch is stopped. **Note:** This condition indicates that a pool of the mapper is full but the entire mapper may not be.
Tablewalk duration	This signal is asserted every cycle when a tablewalk is active. While a tablewalk is active, any request attempting to access the TLB is rejected and retried.
L1 D-cache entries invalidated from L2	A D-cache invalidated was received from the L2 because a line in L2 was castout.
Out of streams	A new prefetch stream was detected, but no more stream entries were available.
D-SLB miss	An SLB miss for a data request occurred. When there is a miss in the SLB, the operating system must reload the buffer with the information needed for a hit so that the transaction can proceed. Therefore, an SLB miss causes an interrupt (trap) to indicate to the operating system that it needs to resolve the problem.
D-TLB miss	A TLB miss for a data request occurred. Requests that miss the TLB may be retried until the instruction is in the next-to-complete group (unless HID4 is set to allow speculative tablewalks). This may result in multiple TLB misses for the same instruction.
Duration MSR(EE) equals '0'	The ISU sends the MSR[EE] bit to the PMU. It is up to the performance monitor to count the cycles while this bit is '0'.
MSR(EE) equals '0' and interrupt pending	The ISU sends the MSR[EE] bit and a signal indicating that an interrupt is pending to the PMU. It is up to the performance monitor to count the cycles while MSR[EE] equals '0' and the interrupt is pending.
FPR mapper full	The ISU sends a signal indicating that the FPR mapper cannot accept any more groups. Dispatch is stopped. **Note:** This condition indicates that a pool of mappers is full but the entire mapper may not be.
FPU0 add, mult, sub, compare, fsel	This signal is active for one cycle when FPU0 is executing an add, mult, sub, compare, or fsel kind of instruction. The instruction could be **fadd***, **fmul***, **fsub***, **fcmp****, or **fsel**.
FPU0 denormalized operand	This signal is active for one cycle when one of the operands is denormalized.
FPU0 divide	This signal is active for one cycle at the end of the microcode executed when FPU0 is executing a divide instruction. The instruction could be **ffdiv**, **fdivs**, **fdiv.**, or **fdivs.**
FPU0 estimate	This signal is active for one cycle when FPU0 is executing one of the estimate instructions. The instruction could be **fres*** or **frsqrte*** where **xyz*** means **xyz** or **xyz.**
FPU0 finished and produced a result	This signal only indicates finish, not completion.
FPU0 mult-add	This signal is active for one cycle when FPU0 is executing a multiply-add kind of instruction. The instruction could be **fmadd***, **fnmadd***, **fmsub***, or **fnmsub*** where **xyz*** means **xyz**, **xyzs**, **xyz.**, **xyzs.**

Table 10-17. Detailed Event Descriptions (Page 2 of 8)

Event Description	Detailed Description
FPU0 move, estimate	This signal is active for one cycle when FPU0 is executing a move kind of instruction or one of the estimate instructions. The instruction could be **fmr***, **fneg***, **fabs***, **fnabs***, **fres***, or **frsqrte*** where **xyz*** means **xyz** or **xyz**.
FPU0 FPSCR	This signal is active for one cycle when FPU0 is executing an FPSCR move-related instruction. The instruction could be **mtfsfi***, **mtfsb0***, **mtfsb1***, **mffs***, **mtfsf***, or **mcrsf*** where **xyz*** means **xyz**, **xyzs**, **xyz.**, or **xyzs.**.
FPU0 round, convert	This signal is active for one cycle when FPU0 is executing **frsp** or a convert kind of instruction. The instruction could be **frsp***, **fcfid***, or **fcti*** where **xyz*** means **xyz**, **xyzs**, **xyz.**, or **xyzs.**.
FPU0 square root	This signal is active for one cycle at the end of the microcode executed when FPU0 is executing a square root instruction. The instruction could be **fsqrt*** where **xyz*** means **xyz**, **xyzs**, **xyz.**, **xyzs.**.
FPU0 issue queue full	The issue queue for FPU 0 cannot accept any more instructions. Issue is stopped.
FPU0 single precision	This signal is active for one cycle when FPU0 is executing a single-precision instruction.
FPU0 stall 3	This signal indicates that FPU0 has generated a stall in pipe 3 due to overflow, underflow, massive cancel, convert to integer (sometimes), or convert from integer (always). This signal is active during the entire duration of the stall.
FPU0 store	This signal is active for one cycle when FPU0 is executing a store instruction.
FPU1 add, mult, sub, compare, fsel	This signal is active for one cycle when FPU1 is executing an add, mult, sub, compare, or fsel kind of instruction. The instruction could be **fadd***, **fmul***, **fsub***, **fcmp****, or **fsel** where **xyz*** means **xyz**, **xyzs**, **xyz.**, **xyzs.** and **xyz**** means **xyzu** and **xyzo**.
FPU1 denormalized operand	This signal is active for one cycle when one of the operands is denormalized.
FPU1 divide	This signal is active for one cycle at the end of the microcode executed when FPU1 is executing a divide instruction. The instruction could be **fdiv**, **fdivs**, **fdiv.**, or **fdivs.**
FPU1 estimate	This signal is active for one cycle when FPU1 is executing one of the estimate instructions. The instruction could be **fres*** or **frsqrte*** where **xyz*** means **xyz** or **xyz.** .
FPU1 finished and produced a result	This signal only indicates finish, not completion.
FPU1 mult-add	This signal is active for one cycle when FPU1 is executing a multiply-add kind of instruction. The instruction could be **fmadd***, **fnmadd***, **fmsub***, or **fnmsub*** where **xyz*** means **xyz**, **xyzs**, **xyz.**, and **xyzs.**.
FPU1 move, estimate	This signal is active for one cycle when FPU1 is executing a move kind of instruction or one of the estimate instructions. The instruction could be **fmr***, **fneg***, **fabs***, **fnabs***, **fres***, or **frsqrte*** where **xyz*** means **xyz** or **xyz.**.
FPU1 round, convert	This signal is active for one cycle when FPU1 is executing **frsp** or convert kind of instruction. The instruction could be **frsp***, **fcfid***, or **fcti*** where **xyz*** means **xyz**, **xyzs**, **xyz.**, **xyzs.**.
FPU1 square root	This signal is active for one cycle at the end of the microcode executed when FPU1 is executing a square root instruction. The instruction could be **fsqrt*** where **xyz*** means **xyz**, **xyzs**, **xyz.**, **xyzs.**.
FPU1 issue queue full	The issue queue for FPU 1 cannot accept any more instructions. Issue is stopped.
FPU1 single precision	This signal is active for one cycle when FPU1 is executing a single-precision instruction.
FPU1 stall 3	This signal indicates that FPU1 has generated a stall in pipe 3 due to overflow, underflow, massive cancel, convert to integer (sometimes), or convert from integer (always). This signal is active during the entire duration of the stall.
FPU1 store	This signal is active for one cycle when FPU1 is executing a store instruction.
FXU0/LSU0 issue queue full	The issue queue for FXU/LSU unit 0 cannot accept any more instructions. Issue is stopped.
FXU1/LSU1 issue queue full	The issue queue for FXU/LSU 1 cannot accept any more instructions. Issue is stopped.
FXU0 produced a result	The FXU0 finished an instruction and produced a result.

Table 10-17. Detailed Event Descriptions (Page 3 of 8)

Event Description	Detailed Description
FXU1 produced a result	The FXU1 finished an instruction and produced a result.
GPR mapper full	The ISU sends a signal indicating that the GPR mapper cannot accept any more groups. Dispatch is stopped. **Note:** This condition indicates that a pool of mapper is full but the entire mapper may not be.
Dispatch blocked by scoreboard	The ISU sends a signal indicating that dispatch is blocked by the scoreboard.
Dispatch reject	Dispatch successful equals *dispatch_valid* and one cycle later ~*dispatch_reject*.
Dispatch valid	The ISU sends *dispatch_valid* and *dispatch_reject* signals to the PMU. It is up to the performance monitor to look at these signals to count the number of dispatch groups.
Instruction prefetch installed in prefetch buffer	This signal is asserted when a prefetch buffer entry (line) is allocated but the request is not a demand fetch.
Instruction prefetch request	Asserted when a non-canceled prefetch is made to the CIU.
Translation written to I-ERAT	This signal is asserted each time the I-ERAT is written. This indicates that an ERAT miss has been serviced. ERAT misses will initiate a sequence resulting in the ERAT being written. ERAT misses that are later ignored will not be counted unless the ERAT is written before the instruction stream is changed. This should be a fairly accurate count of ERAT missed (best available).
Instructions dispatched count	The ISU sends the number of instructions dispatched.
Valid instruction available	Asserted each cycle when the IFU sends at least one instruction to the IDU.
I-SLB miss	An SLB miss for an instruction fetch has occurred.
I-TLB miss	A TLB miss for an Instruction fetch has occurred.
L1 reload data source valid	The data source information is valid.
L1 prefetches	A request to prefetch data into the L1 was made.
L2 Prefetch	A request to prefetch data into the L2 was made.
larx executed side 0	An **larx** (**lwarx** or **ldarx**) was executed on side 0 (there is no corresponding unit 1 event because **larx** instructions can only execute on unit 0).
L1 D-cache load miss side 0	A load, executing on unit 0, missed the D-cache.
L1 D-cache store side 1	A store executed on unit 1.
L1 D-cache load miss side 1	A load, executing on unit 1, missed the D-cache.
L1 D-cache load side 0	A load executed on unit 0.
LR/CTR mapper full	The ISU sends a signal indicating that the LR/CTR mapper cannot accept any more groups. Dispatch is stopped. **Note:** This condition indicates that a pool of the mapper is full but the entire mapper may not be.
LMQ full	The LMQ was full.
LMQ LHR merge	A D-cache miss occurred for the same real cache line address as an earlier request already in the load miss queue and was merged into the LMQ entry.
LMQ slot 0 allocated	The first entry in the LMQ was allocated.
LMQ slot 0 valid	This signal is asserted every cycle when the first entry in the LMQ is valid. The LMQ has eight entries that are allocated on a FIFO basis.
LRQ full	The ISU sends this signal when the LRQ is full.
LRQ slot 0 allocated	LRQ slot zero was allocated.
LRQ slot 0 valid	This signal is asserted every cycle that slot zero of the store request queue is valid. The SRQ is 32 entries long and is allocated on a round-robin basis.

Table 10-17. Detailed Event Descriptions (Page 4 of 8)

Event Description	Detailed Description
SRQ full	The ISU sends this signal when the SRQ is full.
SRQ slot 0 allocated	SRQ slot zero was allocated.
SRQ slot 0 valid	This signal is asserted every cycle that slot zero of the store request queue is valid. The SRQ is 32 entries long and is allocated round-robin.
SRQ sync duration	This signal is asserted every cycle when a sync is in the SRQ.
LSU busy side 0	LSU 0 is busy rejecting instructions.
D-ERAT miss side 0	A data request (load or store) from LSU 0 missed the ERAT. Requests that miss the D-ERAT are rejected and retried until the request hits in the ERAT. This may result in multiple ERAT misses for the same instruction.
Flush from LRQ SHL, LHL side 0	A load was flushed by unit 1 because a a younger load executed before an older store executed and they had overlapping data. Alternatively, two loads executed out-of-order, they had byte overlap, and there was a snoop in between to an overlapped byte.
Flush SRQ LHS side 0	A store was flushed because a younger load hits an older store that is already in the SRQ or in the same group.
Flush unaligned load side 0	A load was flushed from unit 1 because it was unaligned (crossed a 64-byte boundary, or a 32-byte boundary if it missed the L1).
Flush unaligned store side 0	A store was flushed from unit 1 because it was unaligned.
Floating point load side 0	A floating-point load was executed from LSU unit 0.
SRQ store forwarding side 0	Data from a store instruction was forwarded to a load on unit 0.
LSU busy side 1	LSU 0 is busy rejecting instructions.
D-ERAT miss side 1	A data request (load or store) from LSU 1 missed the ERAT. Requests that miss the D-ERAT are rejected and retried until the request hits in the ERAT. This may result in multiple ERAT misses for the same instruction.
Flush from LRQ SHL, LHL side 1	A load was flushed by unit 1 because a a younger load executed before an older store executed and they had overlapping data. Alternatively, two loads executed out-of-order, they had byte overlap, and there was a snoop in between to an overlapped byte.
Flush SRQ LHS side 1	A store was flushed because younger load hits an older store that is already in the SRQ or in the same group.
Flush unaligned load side 1	A load was flushed from unit 1 because it was unaligned (crossed a 64-byte boundary, or a 32-byte boundary if it missed the L1).
Flush unaligned store side 1	A store was flushed from unit 1 because it was unaligned (crossed a 4KB boundary).
Floating point load side 1	A floating-point load was executed from LSU 1.
Marked IMR reload	A Data L1 Cache reload occurred due to marked load.
Marked L1 reload data source valid	The source information is valid and is for a marked load.
Marked L1 D-cache load miss side 0	A marked load, executing on unit 0, missed the D-cache.
Marked L1 D-cache load miss side 1	A marked load, executing on unit 1, missed the D-cache.
Marked SRQ valid	This signal is asserted every cycle when a marked request is resident in the store request queue.
Marked flush from LRQ SHL, LHL side 0	A marked load was flushed by unit 0 because a a younger load executed before an older store executed and they had overlapping data. Alternatively, two loads executed out-of-order, they have byte overlap, and there was a snoop in between to an overlapped byte.
Marked flush SRQ LHS side 0	A marked store was flushed because a younger load hits an older store that is already in the SRQ or in the same group.
Marked flush unaligned store side 0	A marked store was flushed from unit 0 because it was unaligned.

Table 10-17. Detailed Event Descriptions (Page 5 of 8)

Event Description	Detailed Description
Marked flush unaligned load side 0	A marked load was flushed from unit 0 because it was unaligned (crossed a 64-byte boundary, or a 32-byte boundary if it missed the L1).
LSU side 0 finished IMR	LSU unit 0 finished a marked instruction.
Marked flush from LRQ SHL, LHL side 1	A marked load was flushed by unit 1 because a a younger load executed before an older store executed and they had overlapping data. Alternatively, two loads executed out-of-order, they had byte overlap, and there was a snoop in between to an overlapped byte.
Marked flush SRQ LHS side 1	A marked load was flushed by unit 1 because a a younger load executed before an older store executed and they had overlapping data. Alternatively, two loads executed out-of-order, they had byte overlap, and there was a snoop in between to an overlapped byte.
Marked flush unaligned load side 1	A marked load was flushed from unit 1 because it was unaligned (crossed a 64-byte boundary, or a 32-byte boundary if it missed the L1).
Marked flush unaligned store side 1	A marked store was flushed from unit 1 because it was unaligned (crossed a 4KB page boundary).
LSU side 1 finished IMR	LSU unit 1 finished a marked instruction.
Marked L1 D-cache store miss	A marked store missed the D-cache.
Marked **stcx** fail	A marked **stcx** (**stwcx** or **stdcx**) failed.
Snoop **tlbie**	A **tlbie** was snooped from another processor.
L1 D-cache store miss	A store missed the D-cache.
L1 D-cache store miss	A store missed the D-cache.
L1 D-cache store side 0	A store executed on Unit 0.
L1 D-cache load side 1	A load executed on Unit 1.
stcx failed	An **stcx** (**stwcx** or **stdcx**) failed.
stcx passed	An **stcx** (**stwcx** or **stdcx**) instruction was successful.
XER mapper full	The ISU sends a signal indicating that the XER mapper cannot accept any more groups. Dispatch is stopped. **Note:** This condition indicates that a pool of the mapper is full but the entire mapper may not be.
No instructions fetched	No instructions were fetched this cycle (due to IFU hold, redirect, or I-cache miss).
One or more PowerPC instruction completed	A group containing at least one PowerPC instruction completed. For microcoded instructions that span multiple groups, this will only occur once.
BR issue queue full	The ISU sends a signal indicating that the issue queue that feeds the IFU BR unit cannot accept any more groups (the queue is full of groups).
CR issue queue full	The ISU sends a signal indicating that the issue queue that feeds the IFU CR unit cannot accept any more groups (the queue is full of groups).
Processor cycles	Processor cycles.
Data loaded from L2	DL1 was reloaded from the local L2 due to a demand load.
Data loaded from memory	DL1 was reloaded from memory due to a demand load.
New stream allocated	A new prefetch stream was allocated.
External interrupts	An external interrupt occurred.
FPU executed add	This signal is active for one cycle when FPU0 is executing an add, mult, sub, compare, or fsel kind of instruction. The instruction could be **fadd***, **fmul***, **fsub***, **fcmp****, or **fsel** where **xyz*** means **xyz**, **xyzs**, **xyz.**, **xyzs.** and **xyz**** means **xyzu** and **xyzo.** Combined Unit 0 + Unit 1.

Table 10-17. Detailed Event Descriptions (Page 6 of 8)

Event Description	Detailed Description
FPU received denormalized data	This signal is active for one cycle when one of the operands is denormalized. Combined Unit 0 + Unit 1.
FPU executed FDIV instruction	This signal is active for one cycle at the end of the microcode executed when FPU0 is executing a divide instruction. The instruction could be **fdiv**, **fdivs**, **fdiv**. **fdivs**. Combined Unit 0 + Unit 1.
FPU executed FEST instruction	This signal is active for one cycle when executing one of the estimate instructions. The instruction could be **fres*** or **frsqrte*** where **xyz*** means **xyz** or **xyz.** Combined Unit 0 + Unit 1.
FPU produced a result	FPU finished and produced a result. This only indicates finish, not completion. Combined Unit 0 + Unit 1.
FPU executed multiply-add instruction	This signal is active for one cycle when FPU0 is executing a multiply-add kind of instruction. The instruction could be **fmadd***, **fnmadd***, **fmsub***, or **fnmsub*** where **xyz*** means **xyz**, **xyzs**, **xyz.**, **xyzs.** Combined Unit 0 + Unit 1.
FPU executing FMOV or FEST instructions	This signal is active for one cycle when executing a move kind of instruction or one of the estimate instructions. The instruction could be **fmr***, **fneg***, **fabs***, **fnabs***, **fres***, or **frsqrte*** where **xyz*** means **xyz** or **xyz.** Combined Unit 0 + Unit 1.
FPU executed FRSP or FCONV instructions	This signal is active for one cycle when executing **frsp** or a convert kind of instruction. The instruction could be **frsp***, **fcfid***, **fcti*** where **xyz*** means **xyz**, **xyzs**, **xyz.**, **xyzs.** Combined Unit 0 + Unit 1.
FPU executed FSQRT instruction	This signal is active for one cycle at the end of the microcode executed when FPU0 is executing a square root instruction. The instruction could be **fsqrt*** where **xyz*** means **xyz**, **xyzs**, **xyz.**, **xyzs.** Combined Unit 0 + Unit 1.
Cycles FPU issue queue full	Cycles when one or both FPU issue queues are full.
FPU executed single-precision instruction	FPU is executing a single-precision instruction. Combined Unit 0 + Unit 1.
FPU stalled in pipe 3	FPU has generated a stall in pipe 3 due to overflow, underflow, massive cancel, convert to integer (sometimes), or convert from integer (always). This signal is active during the entire duration of the stall. Combined Unit 0 + Unit 1.
FPU executed store instruction	FPU is executing a store instruction. Combined Unit 0 + Unit 1.
Cycles FXLS queue is full	Cycles when one or both FXU/LSU issue queues are full.
FXU busy	FXU0 and FXU1 are both busy.
FXU produced a result	The fixed-point unit (Unit 0 + Unit 1) finished a marked instruction. Instructions that finish may not necessarily complete.
FXU idle	FXU0 and FXU1 are both busy.
FXU0 busy FXU1 idle	FXU0 is busy while FXU1 is idle.
FXU1 busy FXU0 idle	FXU0 is idle while FXU1 is busy.
Cycles GCT empty	The global completion table is completely empty.
Completion table full	The ISU sends a signal indicating that the GCT is full.
Group completed	A group completed. Microcoded instructions that span multiple groups will generate this event once per group.
Group dispatches	A group was dispatched.
Group dispatch rejected	A group that previously attempted dispatch was rejected.
Group dispatch success	Number of groups successfully dispatched (not rejected).
Group marked in IDU	A group was sampled (marked).
Instructions completed	Number of eligible instructions that completed.

Table 10-17. Detailed Event Descriptions (Page 7 of 8)

Event Description	Detailed Description
Instructions fetched from L1	An instruction fetch group was fetched from L1. Fetch groups can contain up to eight instructions.
Instructions fetched from L2	An instruction fetch group was fetched from L2. Fetch groups can contain up to eight instructions.
Instructions fetched from memory	An instruction fetch group was fetched from memory. Fetch groups can contain up to eight instructions.
Instructions fetched from prefetch	An instruction fetch group was fetched from the prefetch buffer. Fetch groups can contain up to eight instructions.
Cycles is L1 write active	This signal is asserted each cycle a cache write is active.
larx executed	An **larx** (**lwarx** or **ldarx**) was executed. This is the combined count from LSU0 + LSU1, but these instructions only execute on LSU0.
L1 D-cache load misses	Total DL1 load references that miss the DL1.
L1 D-cache load references	Total DL1 load references.
LSU busy	LSU (Unit 0 + Unit 1) is busy rejecting instructions.
D-ERAT misses	Total D-ERAT misses (Unit 0 + Unit 1). Requests that miss the D-ERAT are rejected and retried until the request hits in the ERAT. This may result in multiple ERAT misses for the same instruction.
LRQ flushes	A load was flushed because a younger load executed before an older store executed and they had overlapping data. Alternatively, two loads executed out-of-order, they had byte overlap, and there was a snoop in between to an overlapped byte.
SRQ flushes	A store was flushed because a younger load hits an older store that is already in the SRQ or in the same group.
LRQ unaligned load flushes	A load was flushed because it was unaligned (crossed a 64-byte boundary, or a 32-byte boundary if it missed the L1).
SRQ unaligned store flushes	A store was flushed because it was unaligned.
LSU executed floating-point load instruction	
Cycles LMQ and SRQ empty	Cycles when both the LMQ and SRQ are empty (LSU is idle).
Cycles SRQ empty	The store request queue is empty.
SRQ store forwarding side 1	Data from a store instruction was forwarded to a load on Unit 1.
Marked instruction BRU processing finished	The branch unit finished a marked instruction. Instructions that finish may not necessarily complete.
Marked instruction CRU processing finished	The condition register unit finished a marked instruction. Instructions that finish may not necessarily complete.
Marked data loaded from L2	DL1 was reloaded with modified (M) data from the L2 of a chip on this MCM due to a marked load.
Marked data loaded from memory	DL1 was reloaded with modified (M) data from the L2 of another MCM due to a marked load.
Marked instruction FPU processing finished	One of the floating-point units finished a marked instruction. Instructions that finish may not necessarily complete.
Marked instruction FXU processing finished	One of the fixed-point units finished a marked instruction. Instructions that finish may not necessarily complete.
Marked group completed	A group containing a sampled instruction completed. Microcoded instructions that span multiple groups will generate this event once per group.
Marked group dispatched	A group containing a sampled instruction was dispatched.

Table 10-17. Detailed Event Descriptions (Page 8 of 8)

Event Description	Detailed Description
Marked group issued	A sampled instruction was issued.
Marked group completion timeout	The sampling timeout expired indicating that the previously sampled instruction is no longer in the processor.
Marked instruction finished	One of the execution units finished a marked instruction. Instructions that finish may not necessarily complete.
Marked L1 D-cache load misses	
Marked instruction LSU processing finished	One of the load/store units finished a marked instruction. Instructions that finish may not necessarily complete.
Marked LRQ flushes	A marked load was flushed because a younger load executed before an older store executed and they had overlapping data. Alternatively, two loads executed out-of-order, they have byte overlap, and there was a snoop in between to an overlapped byte.
Marked SRQ flushes	A marked store was flushed because a younger load hits an older store that is already in the SRQ or in the same group.
Marked unaligned load flushes	A marked load was flushed because it was unaligned (crossed a 64-byte boundary, or a 32-byte boundary if it missed the L1).
Marked unaligned store flushes	A marked store was flushed because it was unaligned.
Marked store instruction completed	A sampled store has completed (data home).
Marked store completed with intervention	A marked store previously sent to the memory subsystem completed (data home) after requiring intervention.
Marked store sent to storage subsystem	A sampled store has been sent to the memory subsystem.
Run cycles	Processor cycles gated by the run latch.
L1 D-cache store references	Total DL1 store references.
Completion stopped	The RAS unit has signaled completion to stop.
Time-base bit transition	Occurs when the selected time-base bit (as specified in MMCR0[TBSEL]) transitions from '0' to '1'.
Threshold timeout	The threshold timer expired.
Work held	The RAS unit has signaled completion to stop and there are groups waiting to complete.

11. System Design

11.1 I2C Interface

I2C (Interconnect for Integrated Circuits) is a standard bus developed by Philips Electronics.[1] The I2C Slave in the 970MP converts data sent across an I2C bus into native JTAG commands. The I2C slave can be used as a test access port (TAP) controller that interfaces with the Access macro or with other IEEE 1149.1 compatible devices in order to read, write, and scan registers within a chip.

11.2 Bus Initialization, Configuration, Power Management, and Test

11.2.1 Bus Initialization

The bus devices use a physical layer initialization sequence to initialize the bus. A specific pattern is sent across the bus, which initializes the processor interface in slave devices. This sequence is described in *Section 11.3.1 Initialization at Power-On Reset* on page 379.

11.2.2 Configurable Parameters

The processor interconnect defines multiple configurable parameters for efficient operation of the bus. The values that can be programmed into these parameter registers are technology and implementation-dependent. During the initialization process at system start-up, the power-management unit identifies the system configuration and programs the individual devices attached to the bus (that is, the North Bridge and the processors) with the appropriate values using the I2C device interfaces. All values are in bus beats. For parameters that cross the processor interface, the values are from the final locally clocked flip-flop or latch output to the first locally clocked flip-flop or latch input, after deskewing has taken place through the processor interconnect.

Figure 11-1 shows the configurable timing parameters, COMPACE and STATLAT. COMPACE is the minimum number of bus beats between command packets issued from the processor. STATLAT is the number of bus beats between the last beat of the address/data (AD) packet and the first beat of the transfer-handshake (TH) packet.

1. I2C standard (IIC) for a serial bus. For more information see: http://www-us2.semiconductors.philips.com/i2c/.

Figure 11-1. Configurable Timing Parameters

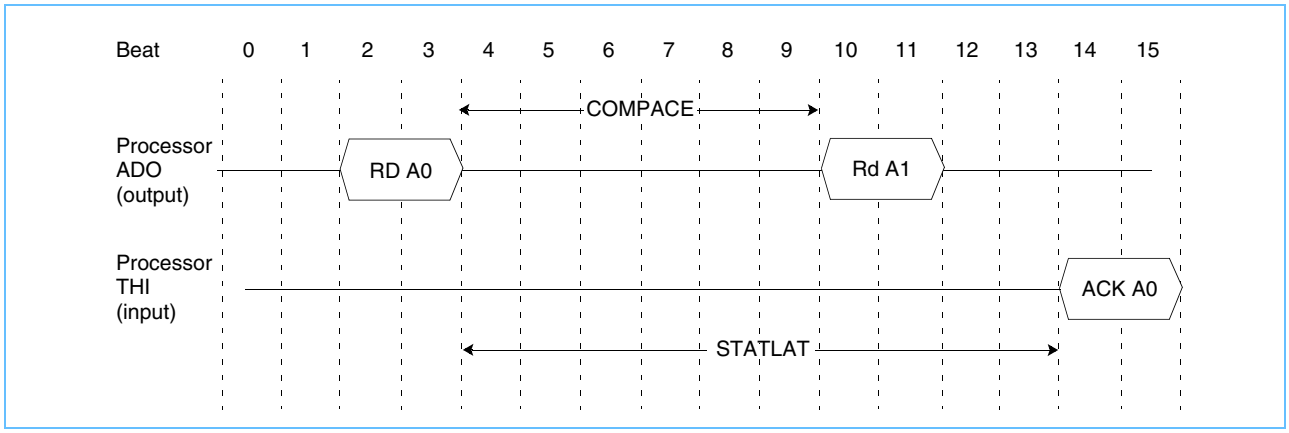


Figure 11-2 shows the North Bridge configurable timing parameters, SNOOPWIN, SNOOPLAT, and PAAMWIN. SNOOPWIN is the minimum number of idle bus beats between reflected command packets. PAAMWIN is the minimum number of bus beats between command packets reflected from the North Bridge to the processors when there is an address collision (shown as A0 in *Figure 11-2*). SNOOPLAT is the number of bus beats between the last beat of a reflected command packet to the first beat of the individual snoop responses from each of the processors received at the North Bridge. SNOOPLAT includes the time of flight across the interface and any switch devices interposed between the North Bridge and a processor.

Figure 11-2. North Bridge Configurable Timing Parameters

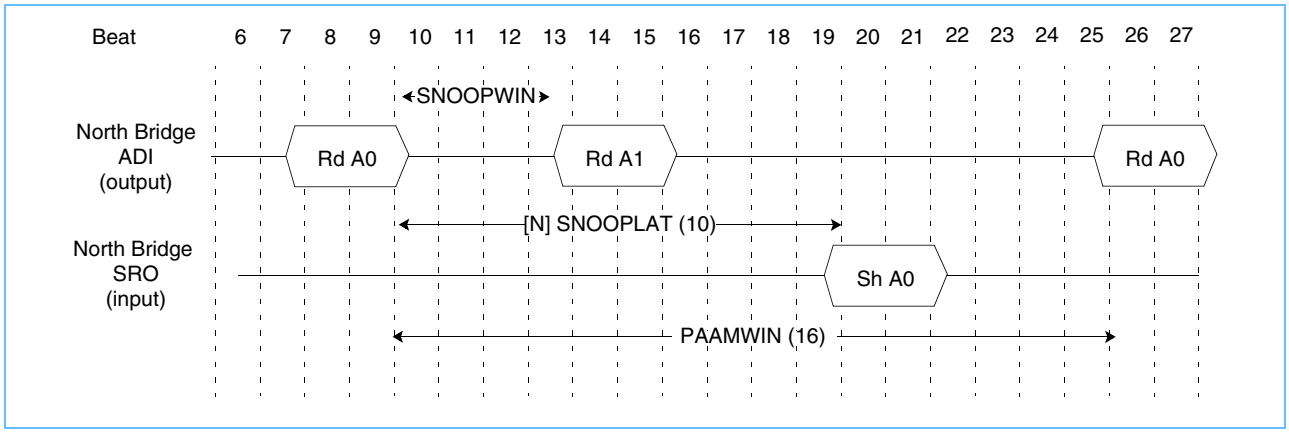


Figure 11-3 shows the processor configurable timing parameters, SNOOPLAT and SNOOPACC. SNOOPLAT is defined above. SNOOPACC is the number of bus beats between the last beat of the individual snoop response sent from a processor to the first beat of the accumulated snoop response received from the North Bridge. SNOOPACC includes the time of flight across the interface and any switch devices interposed between a processor and the North Bridge.

Figure 11-3. Processor Configurable Timing Parameters

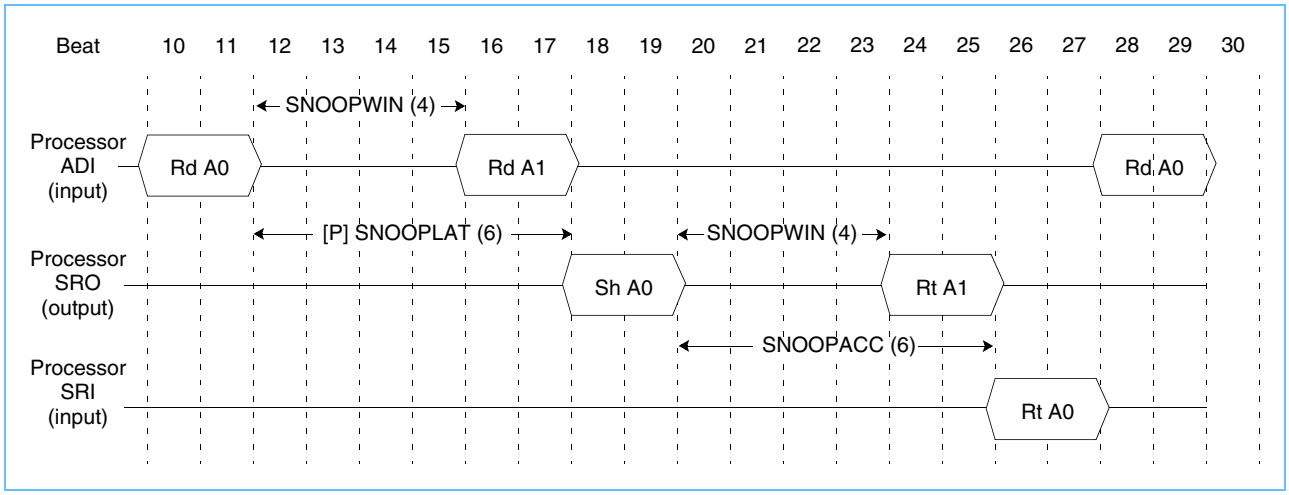


Table 11-1. Programmable Delay Parameters

Parameter	Processor	North Bridge	Range in Bus Beats			Description
			Minimum	Typical	Maximum	
COMPACE	Y	N	2	—	14	Command pipeline delay. See *Section 8.2.1.4 Command Pacing* on page 241.
STATLAT	Y	N	4	—	30	Transfer-handshake response latency. See *Section 8.2.3 Transfer-Handshake Packets* on page 245 and *Section 8.4.2 Memory Read Transactions (General)* on page 254.
STATLAT	N	Y	—	22	—	
SNOOPWIN	N	Y	—	4	—	Snoop window pacing. See *Section 8.3.1 Snoop-Response Bus Implementation* on page 249 and *Section 8.4.2 Memory Read Transactions (General)* on page 254.
SNOOPLAT	N	Y	—	25	—	North Bridge snoop latency. See *Section 8.3.1 Snoop-Response Bus Implementation* on page 249.
SNOOPLAT	Y	N	6	6	12	Processor snoop latency. See *Section 8.3.1 Snoop-Response Bus Implementation* on page 249.
SNOOPACC	Y	N	9	—	24	North Bridge snoop accumulation delay. See *Section 8.3.1 Snoop-Response Bus Implementation* on page 249 and *Section 8.4.2.3 Read with Intent to Modify Burst Transaction* on page 256.

11.2.3 Configuration Interface

An I^2C interface is used by the power-management unit to configure the processor interconnect bus parameters. The interface complies with the *I^2C Bus Specification*. *Table 11-2* lists the I^2C interface signals. *Table 11-3* lists the I^2C registers, which are described in this section.

The I^2C interface consists of two bidirectional signals, $\overline{\text{I2CCK}}$ and $\overline{\text{I2CDT}}$. Both signals use open-drain drivers that require external pull-up resistors. Multiple devices can be connected to the same signals. The PROCID[0:1] inputs are used to address a specific processor.

Table 11-2. I^2C Interface Signals

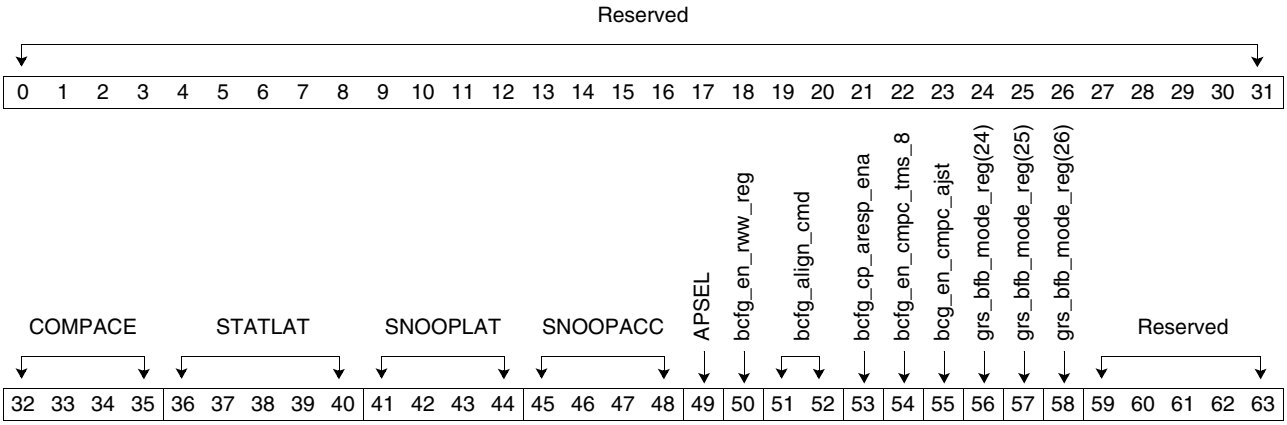
Signal	Polarity	Name
$\overline{\text{I2CCK}}$	Active Low	I^2C interface clock input
$\overline{\text{I2CDT}}$	Active Low	I^2C interface data input/output
PROCID[0:1]	Active High	Processor identification input

Table 11-3. I^2C Registers Used by the 970MP Processor Interconnect

Name	Mode	Address	Description	See Page
PI Status Register	Read	x'084001'	Processor Interconnect Status Register	440
PI Mode Register 0	Read/Write	x'083000'	Processor Interconnect Mode Registers 0	436
PI Mode Register 1	Read/Write	x'083100'	Processor Interconnect Mode Registers 1	437
PI Mode Register 2	Read/Write	x'083200'	Processor Interconnect Mode Registers 2	438
PI Mode Register 3	Read/Write	x'083300'	Processor Interconnect Mode Registers 3	439
BUSCONF	Read/Write	x'0A8000'	Processor Configurable Timing Delay Parameter Register	373

11.2.3.1 Processor Configurable Timing Delay Parameter Register (BUSCONF)

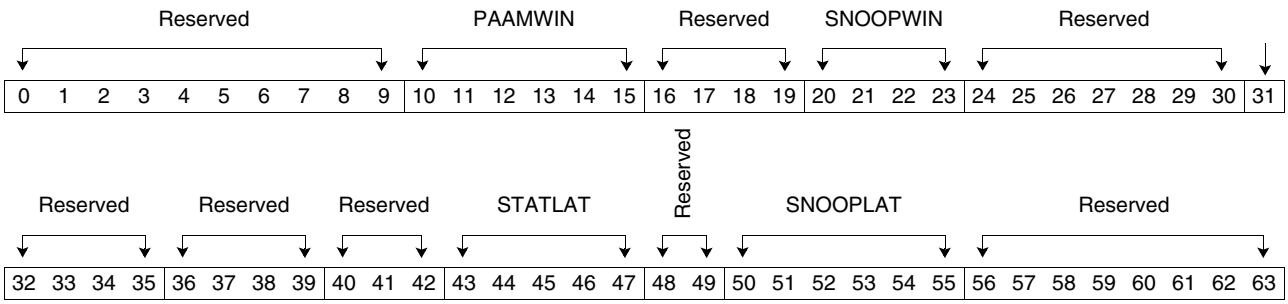
SCOM Address x'0A8000'



Bits	Field Name	Description
0:31	—	Reserved.
32:35	COMPACE	Command pipeline delay.
36:40	STATLAT	Transfer-handshake response latency.
41:44	SNOOPLAT	Processor snoop latency.
45:48	SNOOPACC	Snoop accumulation delay.
49	APSEL	Bus encode disable. ('1' = disable)
50	bcfg_en_rww_reg	Enable commands during data writes. ('0' = enable)
51:52	bcfg_align_cmd	Align command. 00 Command out on even or odd. 01 Command out on even. 10 Command out on odd.
53	bcfg_cp_aresp_ena	Disables wait for cresp for castouts and pushes. ('1' = disable)
54	bcfg_en_cmpc_tms_8	Enables a longer wait period before "back-off."
55	bcg_en_cmpc_ajst	Enables bus "back-off" of sending command.
56	grs_bfb_mode_reg(24)	Sets bus mode to no encode with single error correct and some double error detect.
57	grs_bfb_mode_reg(25)	Sets bus mode to no encode with single error correct and double error detect.
58	grs_bfb_mode_reg(26)	Power tuning engine disable.
59:63	—	Reserved.

11.2.3.2 North Bridge Configurable Timing Delay Parameter Register

SCOM Address x'0A8000'



Bits	Field Name	Description
0:9	—	Reserved.
10:15	PAAMWIN	Minimum number of bus beats between command packets reflected from the North Bridge to the processors when there is an address collision.
16:19	—	Reserved.
20:23	SNOOPWIN	Snoop window pacing.
24:30	—	Reserved.
31		Bus encode disable. 1 Disabled.
32:35	—	Reserved.
36:39	—	Reserved.
40:42	—	Reserved.
43:47	STATLAT	Transfer-handshake response latency.
48:49	—	Reserved.
50:55	SNOOPLAT	North Bridge snoop latency.
56:63	—	Reserved.

11.2.4 Power Management

The processor interconnect participates in the system power management through two asynchronous control signals called quiescent request (QREQ) and quiescent acknowledgment (QACK). QREQ is a processor output signal that is asynchronously sampled by the local clock of the North Bridge. QACK is a North Bridge output signal that is asynchronously sampled by the local clock of the processor and other bus masters.

Figure 11-4 on page 376 shows the sequence of steps for the processor to enter Doze or Nap mode. *Figure 11-5* on page 377 shows the sequence of complementary steps taken by the North Bridge in response to the assertion or negation of QREQ by a processor. In Doze mode, the processor must be capable of snooping all reflected command packets from the North Bridge. In Nap mode, the processor is not required to snoop transactions, although it must be capable of returning to Doze mode for the purpose of snooping if QACK is negated.

In the normal (or desired) sequence of events, the processor and North Bridge observe a 4-phase handshake for QREQ and QACK. The processor first asserts QREQ after the processor has quiesced, the snoopers are idle, and all outstanding processor interconnect bus transactions have completed. The processor then waits for the North Bridge to assert QACK. While the processor is waiting for the assertion of QACK, it is in an intermediate mode called Doze. Once the North Bridge asserts QACK, the processor enters Nap mode. To exit Nap mode, the processor negates QREQ and then waits for the North Bridge to negate QACK before returning to the Run state.

There are a few scenarios in which the 4-phase handshake is preempted.

1. While in Doze mode, the North Bridge reflects command packet snooping. The action taken by the processor is to negate QREQ while snooping the reflected command packet and while staying in Doze mode.
2. While in Doze mode, the processor receives an interrupt. The action taken by the processor is to negate QREQ and return to the Run state.
3. While in Nap mode, the North Bridge negates QACK while the processor has QREQ asserted. The processor must then return to Doze mode within 64 bus clocks so that it can return to snooping reflected command packets from the North Bridge.

As shown in *Figure 11-5* on page 377, the North Bridge normally negates QACK when QREQ is negated by any of the attached processors. However, it might also negate QACK if there is bus activity from any of the other attached bus devices that can be a bus master.

Figure 11-4. Processor QREQ and QACK Signalling

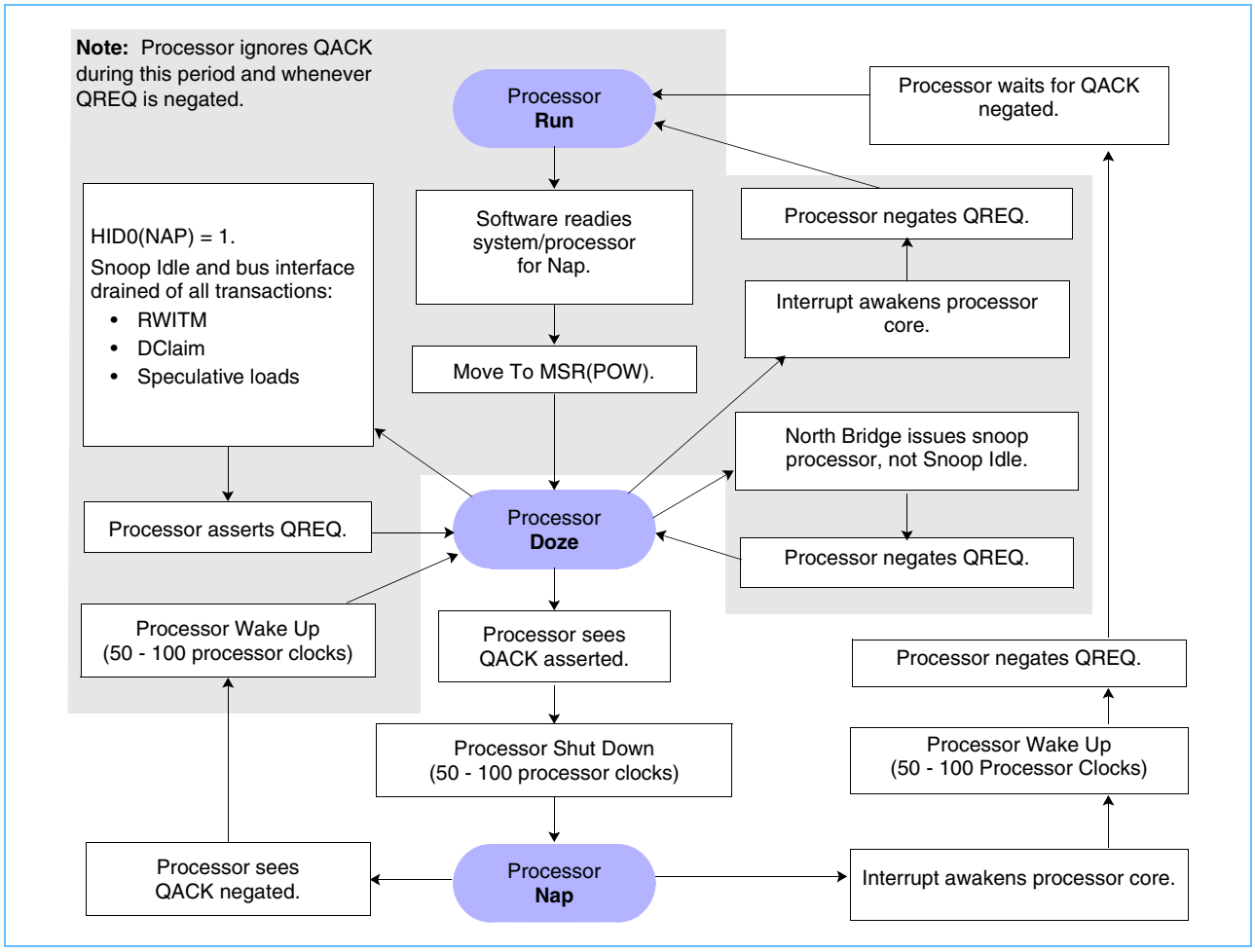
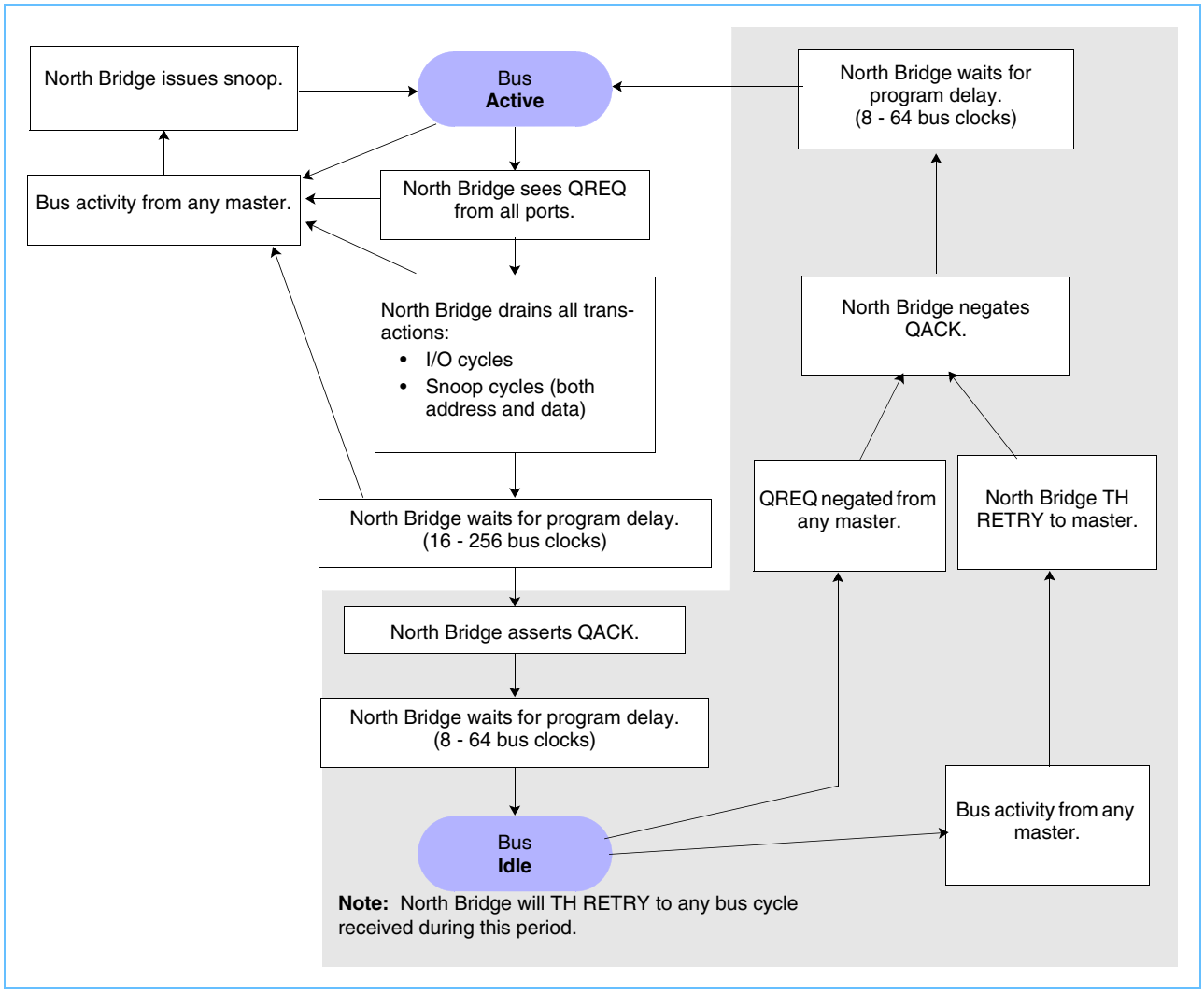


Figure 11-5. North Bridge QREQ and QACK Signalling



11.2.5 Reliability, Availability, and Serviceability (RAS) Requirement

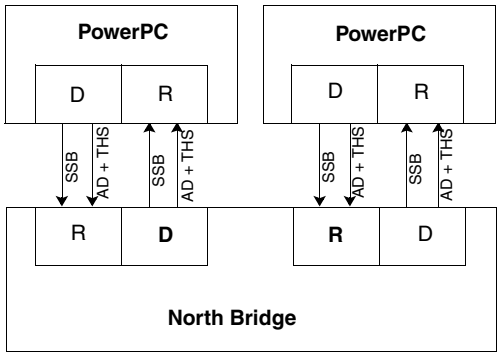
All devices attached to the processor interconnect bus must implement three registers that capture errors and assist in the isolation of failures. RAS circuitry ensures that a processor interconnect implementation that targets a particular processor will remain stable across various target processors.

A Fault Isolation Register (FIR), a Fault Isolation Capture Register (FICR), and a Fault Isolation Mask Register (FIMR) must be implemented. The FIR captures all failures that occur. The register is not frozen on the first error but continues to accumulate all detected errors. The FICR is used to log the first detected error. This register is a masked copy of the FIR (output of the FIR is masked with the FIMR) and is frozen once the first error is detected. The actual errors that are logged and the error reporting mechanism are system-dependent (see *Chapter 4 Exceptions* for more information).

11.3 Processor Interconnect Electrical Interface

The processor interconnect uses high-speed, source-synchronous buses (SSBs) to transfer data between the PowerPC and North Bridge chips, and to support the cache-coherency snooping protocols for multiprocessor configurations. The SSBs are unidirectional point-to-point connections between a drive side (D) and a receive side (R). SSBs are put into pairs to form a bidirectional channel between a PowerPC and a North Bridge chip as shown in *Figure 11-6*.

Figure 11-6. Bus Diagram of a Dual-Processor 970MP Processor Interconnect-Based System



Source-synchronous bus (SSB) data is transferred on every bus clock edge; that is, double the data rate (DDR) of the bus-clock frequency. There are 50 signal lines per SSB. Two lines are used for the differential bus clock lines, 44 signal lines are used to communicate 36 bits of logical data, and four signal lines are used for the differential snoop-response bus. The 36 data bits consist of 35 bits of the address/data (AD) channel and a single bit for the transfer-handshake bus (TH).

The SSBs achieve high-speed operation using low-cost packaging solutions by exploiting four features:

1. **Source-synchronous signalling**. The differential bus clocks are bundled with the single-ended data signals.
2. **Far-end (parallel) termination**. The single-ended data signals use parallel termination at the far end of the signal line to absorb signal reflections and maintain a quasi-constant current loading for each data signal line.
3. **Balanced coding**. The application of balanced coding to the SSB maintains a quasi-constant current loading across the entire SSB interface. Within the SSB, there is no net current flow across the power planes. This dramatically reduces noise problems due to power-supply rail collapse (that is, di/dt noise) and current voltage offsets between the chips.
4. **Point-to-point unidirectional signalling**. Restricting the signal fan-out to a single point and keeping the signal flow unidirectional mitigates problems associated with high-frequency signal attenuation.

11.3.1 Initialization at Power-On Reset

The receive-side circuitry for the SSBs inside a processor interconnect system may require initialization at power-on to deskew data signal lines, align the bus clocks, and synchronize the receive-side FIFO queues to the local clock domains of the ASICs and processors. Within a processor interconnect system, there is the concept of time zero, which is globally established across all the chips. In the processor interconnect, time zero is derived from the phase synchronization (*psync*) and global system clock (SYSCLK) signals (see *Section 11.3.2 Target Cycle* on page 379).

The purpose of the initial alignment pattern (IAP) is to establish the settings for the delay lines of the per bit deskew circuitry and optimize the positioning of the sampling clocks on the receive side. During IAP, each drive side transmits a bit pattern sequence across each SSB. This pattern is repeated by the drive side for as many bit times (for example, 500,000) as needed by the initialization sequential circuitry on the receive side. The I^2C interface controls for how long the pattern is repeated. Upon IAP completion, the receive-side reports its status through a 10-bit PI Status Register, which is accessible from the I^2C interface. An all-zero result stored in PI Status Register indicates that the IAP completed without error. A non-zero pattern indicates that there was an error. *Section PI Status Register* on page 440 describes the bit fields and their meaning.

The sequence of the power-on reset steps is:

1. Stabilize and lock the clocks to the globally distributed SYSCLK.
2. The drive side of each SSB begins transmission of the test patterns for receive-side calibration and optimization. This step is initiated from the I^2C interface by a sequence that is system dependent.

Wait for a completion signal from each receive side SSB that has completed the IAP. The completion signal is registered and can be accessed from the I^2C interface. The location of the register and how it is accessed through the I^2C interface is implementation dependent. The transmission of the test patterns is terminated once the completion signal is detected. The results of the initialization can be read out from the I^2C interface, and the bus is ready for general system use.

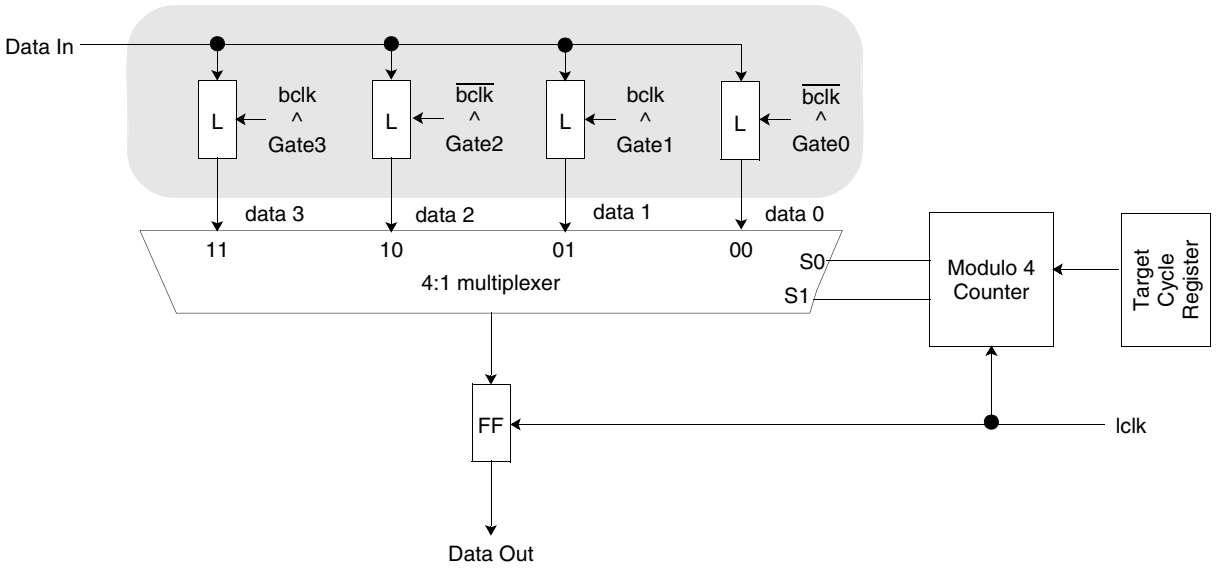
11.3.2 Target Cycle

The flight time of a data signal from the drive side to the receive side of an SSB can extend beyond the period of a single bus clock. The principles of the processor interface allow data and clocks signals to take multiple bus clock cycles to travel from one side to the other.

Furthermore, each receive side can be programmed to transfer SSB data across the time-domain boundary on the same target beat relative to time zero, which is the globally synchronized time domain for all of the processors in a processor interconnect system.

This synchronization can be accomplished using a FIFO-type circuit such as the one in *Figure 11-7*. The four gate signals (Gate0 through Gate3) are derived from the incoming bus clock (bclk) of the SSB. These signals are half the frequency of the bus clock, have a 50% duty cycle, and are 90 degrees out of phase from each other (see *Figure 11-8* on page 380). During the IAP, the gate signals are shifted one bit at a time until the '1' in the IAP pattern is aligned into the rightmost latch (data 0) and the '0' is captured in the leftmost latch (data 3). This alignment procedure occurs in the shaded box of *Figure 11-7*.

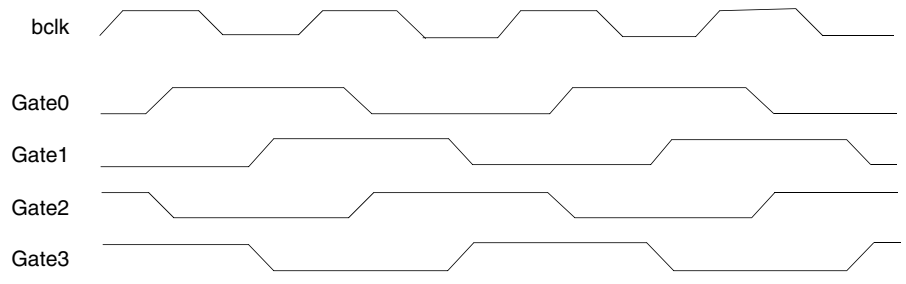
Figure 11-7. Receive-Side FIFO Circuit



The 4:1 multiplexer and the modulo 4 counter are used to cross the time domain from that of the SSB bus clock to the local clock (Lclk) of the chip. A static timing analysis determines the worst case aggregate latency from the drive side through the receive side up to the input point of the data out flip-flop (see *Figure 11-7*).

The combination of the 4:1 multiplexer and modulo 4 counter establishes four possible target cycles for transferring data between the two time domains. The duration of each target cycle equals one bit time. Depending on the results of the worst case analysis, it might be determined that SSB input data cannot be clocked into the data out flip-flop in the same target cycle that it arrives. This will occur if the timing violates the set-up and hold time requirements of the data out flip-flop. In this case, one of the other three target cycles is selected. For example, the following cycle would allow the shortest safe latency, but later cycles would provide larger set-up times. The target cycle is programmed through the I^2C interface by loading a 2-bit value into the Target Cycle Register, which in turns initializes the modulo 4 counter relative to time zero.

Figure 11-8. Timing Diagram Showing Relationship Between Bclk and the Four Gate Signals



11.4 Processor Interconnect Bus Error Detection and Correction

11.4.1 Error Detection for Balanced Encoding

The processor interconnect bus protocol defines an encoding of each beat of information on the inbound address/data (ADI) and outbound address/data (ADO) bus, so that exactly half the signals carry a '1' bit and half the signals carry a '0' bit on each beat. This is done by converting the 36 bits of information on each bus to the 44-bit pattern that is transferred, in a scheme called balanced coding. This balanced coding scheme implicitly provides parity checking of the bus signals, in that an unequal number of ones and zeros in any beat indicates an error. This balanced coding bus mode is selected by setting the BUSCONF bit 49 to '0' (see *Section 11.2.3.1 Processor Configurable Timing Delay Parameter Register (BUSCONF)* on page 373).

11.4.2 Error Detection for Alternative Encodings

The 970MP design supports three unencoded bus modes, in which bits 0:35 of the ADI and ADO bus carry the address and data information, while bits 36:43 carry checking information. Bits 49, 56, and 57 determine the unencoded bus modes as follows:

100 This mode, described in *Section 11.4.2.1 Single-Error and Double-Error Detection*, does not provide single error correction. It only provides single and partial double bit detection.

110 This mode is the same as mode '100' except that single bit errors are corrected.

101 This mode is described in *Section 11.4.2.2 Single-Error Correct, Double-Error Detection*.

111 This mode is undefined.

11.4.2.1 Single-Error and Double-Error Detection

The first of these unencoded bus modes implements the following 10-input parity functions to generate the eight check bits:

```
b36 = P( b0, b6, b7, b29, b30, b31, b32, b33, b34, b35 )
b37 = P( b0, b1, b6, b23, b24, b25, b26, b27, b28, b35 )
b38 = P( b0, b1, b2, b18, b19, b20, b21, b22, b28, b34 )
b39 = P( b1, b2, b3, b14, b15, b16, b17, b22, b27, b33 )
b40 = P( b2, b3, b4, b11, b12, b13, b17, b21, b26, b32 )
b41 = P( b3, b4, b5, b9,  b10, b13, b16, b20, b25, b31 )
b42 = P( b4, b5, b7, b8,  b10, b12, b15, b19, b24, b30 )
b43 = P( b5, b6, b7, b8,  b9,  b11, b14, b18, b23, b29 )
```

where P(0 to 7) computes even parity over its input signals.

In the receiver, the error syndrome is computed by exclusive ORing the received and generated check bits. A syndrome of x'00' results when the received and generated check bits match, indicating that no error occurred. A non-zero syndrome indicates that an error occurred. This check bit implementation will detect any single-bit or double-bit error over the 44-bit pattern. This first unencoded bus mode is selected by setting BUSCONF bits 49, 56, and 57 to '100'.

Note: Single-bit errors that occur using this bus mode yield syndromes that allow the failing bit to be identified. However, some double-bit errors yield those same single-bit error syndromes. For this reason, this mode can be used to detect all single-bit and double-bit errors, but cannot be safely used to correct single-bit errors.

11.4.2.2 Single-Error Correct, Double-Error Detection

The second unencoded bus mode implements the following even parity functions to generate the eight check bits:

```
b36 = P( b23, b24, b25, b26, b27, b28, b29, b30, b31, b32, b33, b34, b35 )
b37 = P( b9,  b10, b11, b12, b13, b14, b15, b16, b17, b18, b19, b20, b21, b22 )
b38 = P( b3,  b4,  b5,  b6,  b7,  b8,  b18, b19, b20, b21, b22, b33, b34, b35 )
b39 = P( b2,  b5,  b6,  b7,  b8,  b15, b16, b17, b22, b29, b30, b31, b32 )
b40 = P( b1,  b3,  b4,  b8,  b12, b13, b14, b17, b21, b26, b27, b28, b32, b35 )
b41 = P( b0,  b1,  b2,  b4,  b7,  b10, b11, b14, b20, b24, b25, b28, b31 )
b42 = P( b0,  b1,  b3,  b6,  b9,  b11, b13, b16, b19, b23, b25, b27, b30, b34 )
b43 = P( b0,  b2,  b5,  b9,  b10, b12, b15, b18, b23, b24, b26, b29, b33 )
```

In the receiver, the error syndrome is computed by exclusive ORing the received and generated check bits. A syndrome of x'00' results when the received and generated check bits match, indicating that no error occurred. A non-zero syndrome indicates that an error occurred. *Table 11-4* on page 383 lists the syndromes from all single-bit errors, along with which failing bit causes that syndrome.

All non-zero syndromes that are not listed in *Table 11-4* indicate double-bit errors.

This check bit implementation can be used to correct any single-bit error and to detect any double-bit error over the 44-bit pattern. This second unencoded bus mode is selected by setting BUSCONF bits 49, 56, and 57 to '101'.

Table 11-4. Bit Error Position Identifier

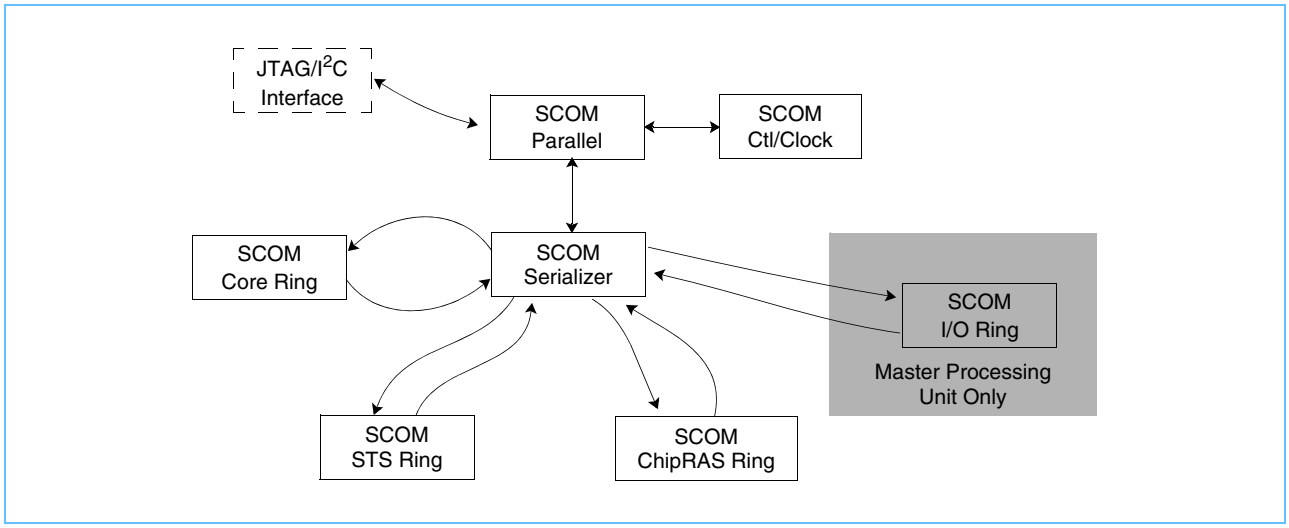
Syndrome	Failing Bit	Syndrome	Failing Bit
x‘07’	0	x‘70’	22
x‘0E’	1	x‘83’	23
x‘15’	2	x‘85’	24
x‘2A’	3	x‘86’	25
x‘2C’	4	x‘89’	26
x‘31’	5	x‘8A’	27
x‘32’	6	x‘8C’	28
x‘34’	7	x‘91’	29
x‘38’	8	x‘92’	30
x‘43’	9	x‘94’	31
x‘45’	10	x‘98’	32
x‘46’	11	x‘A1’	33
x‘49’	12	x‘A2’	34
x‘4A’	13	x‘A8’	35
x‘4C’	14	x‘80’	36
x‘51’	15	x‘40’	37
x‘52’	16	x‘20’	38
x‘58’	17	x‘10’	39
x‘61’	18	x‘08’	40
x‘62’	19	x‘04’	41
x‘64’	20	x‘02’	42
x‘68’	21	x‘01’	43

12. SCOM Interface and Registers

Scan communication (SCOM) is used to access vital chip debug and diagnostic facilities while the chip is running without stopping clocks. It is implemented as address and data serial rings running through the chip to limit the wiring. Each facility has a unique address on this ring, which is used to address it.

On the 970MP, all the SCOM facilities are per processor units except for the I/O SCOM facilities that are only available on the master processing unit (see *Figure 12-1*). The serial ring is split into four independent rings running in the four clock domains, so that a clock stop in the one domain will not break the SCOM. A small number of facilities that control the SCOM configuration and the chip clocks are addressed directly without using the serial ring.

Figure 12-1. Processor Unit SCOM Topology



12.1 Processor Core SCOM SPR Access

Each processor (core) has two special purpose registers (SPRs) used to access the SCOM interface: SCOMC and SCOMD. SCOMC and SCOMD are both 64-bit read/write SPRs and are used for SCOM Control and SCOM Data respectively. The interface is implemented as a direct connection to the parallel-to-serial converter, which handles the arbitration between the core and service processor.

12.1.1 Operating System Protocol to Access SCOM SPRs

In the 970MP, SCOMC and SCOMD are complete operations. They do not require a software protocol in order to function properly except to disable external (asynchronous) interrupts. Software must check the error bits after performing an SCOMC to ensure that the command successfully completed. *Table 12-1* on page 386 outlines a general software protocol for using these registers.

Table 12-1. Operating System Code to Access SCOM

For SCOM READ	For SCOM WRITE
set MSR[EE] = '0' MTSCOMC MFSCOMD MFSCOMC if Error = '1', branch to SCOM error routine set MSR[EE] = '1'	set MSR[EE] = '0' MTSCOMD MTSCOMC MFSCOMC if Error = '1' branch to SCOM error routine set MSR[EE] = '1'

Asynchronous interrupts must be disabled during these blocks. Otherwise, an interrupt could arrive and make the SCOM port busy. If that occurs between the MF and MT instructions that cause the reads and writes, then the SCOM interface out of the core could malfunction (or at least not perform as software intended).

12.1.2 SCOMD Format

The SCOMD is a 64-bit register. The interpretation of the contents of this register is determined by the SCOMC Status and Control bits. It is the source for outgoing data *during* an SCOM write access (SCOMC[RW] = '1' when MTSCOMC is issued). It is the destination for incoming data *after* an SCOMC read access (SCOMC[RW] = '0' after MTSCOMC completes).

12.1.3 SCOMC Format

The SCOMC is divided into four 16-bit fields, as shown in *Table 12-2* and *Figure 12-2*:

Figure 12-2. SCOMC SPR Format

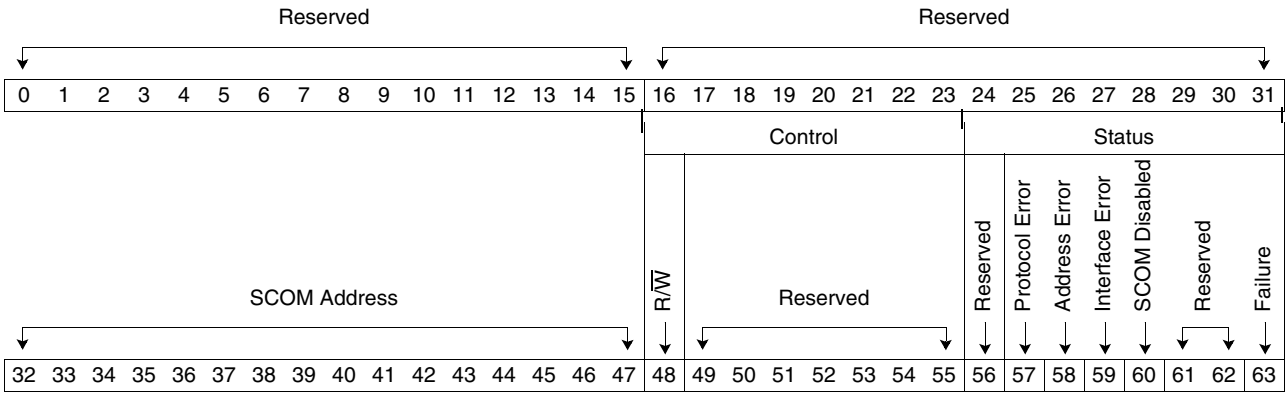


Table 12-2. SCOMC SPR Format (Page 1 of 2)

SCOMC Bits	Type	Usage	Description
0:31	Unused	Reserved	Unused
32:47	Write-Only	Control	SCOM Address (0:15)
48	Write-Only	Control	SCOM Read/Write Request Bit 0 Write request 1 Read request
49:55	Unused	Reserved	Unused control bits

Table 12-2. SCOMC SPR Format (Page 2 of 2)

SCOMC Bits	Type	Usage	Description
56	Read-Only	Reserved	Unused Status bit
57	Read-Only	Status	SCOM Protocol Error
58	Read-Only	Status	SCOM Address Error
59	Read-Only	Status	SCOM Interface Error
60	Read-Only	Status	SCOMC disabled by service processor
61	Read-Only	Status	Reserved (Zero)
62	Read-Only	Status	Reserved (Zero)
63	Read-Only	Status	Failure (SCOMC disabled or Interface Error [formerly Busy])

The reserved fields should be written to zeros by the software on an MTSCOMC and return zeros on an MFSCOMC. The Address and Control fields are undefined while Failure equals '1'.

All SCOMC Status bits will be cleared by the hardware upon an MTSCOMC with the exception of Failure, which is set to indicate to the operating system that the SPR SCOM access is active. Additional status bits will be set depending on the status of the SCOM operation:

- **Protocol Error:** The SCOM hardware has violated a basic protocol, such as giving a grant when not asked or returning a data packet when expecting an address packet. This error bit is *not* cleared on the next MTSCOMC.

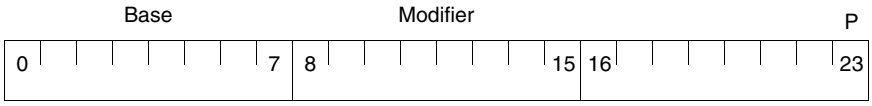
  Note: This bit will probably cause a checkstop to occur and sets a corresponding bit in the Fault Isolation Register (FIR).

  This bit indicates a problem has occurred in the SCOM hardware, and that this interface can no longer be trusted.
- **Address Error:** The SCOM address was not recognized by an SCOM satellite. This indicates that the write did not happen or that the read returned no data (depending on the R/W bit). This error bit is cleared on the next MTSCOMC. This indicates a probable software error.
- **Interface Error:** If the SCOM logic in the arbiter detects an error condition, such as a timeout on the SCOM interface, or if the core hang recovery engages while SCOMC is active, it sends a SCOM reset (screset). This causes the SCOMC operation to be killed and the logic to record an error. The error bit is cleared on the next MTSCOMC and is recoverable (the command must be retried).
- **SCOMC Disabled:** The service processor has the ability to disable a core from becoming an SCOM master, causing the core to treat MTSCOMC as a NOP. MFSCOMC will set this bit, along with Failure and Interface Error to ensure the software realizes this condition.
- **Failure:** Summary indicating if there were any errors since the last MTSCOMC. Formerly the "Busy" bit, which indicated if the SCOMC interface was in use.

12.2 SCOM Address Allocation

Scan communications support a 23-bit address; the 24th bit is a parity bit. This is the address that would be sent to SCOM through the JTAG/I^2C port. The internal SCOM bus, the part that is serialized, needs no more than 16 bits of addressing. So, to simplify the logic, addresses sent internally are truncated to 16 bits. Thus, bits 0 through 15 are sent to the parallel SCOM controller, and bits 16 through 22 must be zero for chip SCOM addresses.

Figure 12-3. Format of an SCOM Address



For the bus interface unit (BIU), each unit's SCOM serial address and data are daisy chained together. For example, the BIU SCOM address and SCOM data out connect to the SCOM address and SCOM data in of the L2 cache. The SCOM address and SCOM data out of the L2 cache connect to the BIU SCOM address and SCOM data in. The BIU SCOM address and SCOM data out connect to the BIU SCOM address and SCOM data in.

Within the BIU, bit 0 of the serial SCOM address is the start bit, and bit 18 is the stop bit. Bit 17 is for SCOM read or write operations ('1' equals read; '0' equals write). Bits 1 through 8 are the SCOM base address for the BIU SCOM registers. Bits 9 through 16 are the actual SCOM address for each register. For the internal serial SCOM data bus, each SCOM register in the BIU is 32 bits wide. Bit 32 of the SCOM data bus is the stop bit.

Figure 12-4. Format of an SCOM Address within the BIU

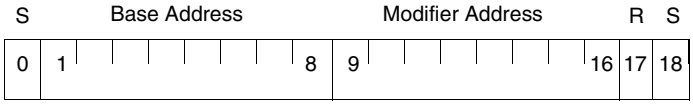


Figure 12-5. Format of an SCOM Data Bus

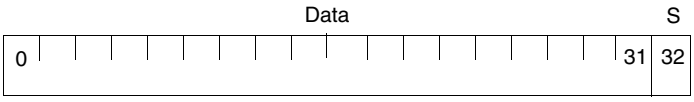


Table 12-3 shows the base address decodes. All data accesses to the SCOM bus are 64 bits wide. The domain column in *Table 12-3* lists the following information for each SCOM register access:

- which clock domain in the chip must be on (use SCOM 800000)
- which SCOM ring must be on (use SCOM 60000). The SCOM ring for the ChipRAS domain is always enabled.

Table 12-3. SCOM Base Addresses

Base SCOM Address (0:7)	Functional Unit	Macro	Domain
x'01'	Core (Debug Unit)	td_cp_dbg	Core
x'02'	Core (RAS Unit)	td_cp_ras	Core
x'03'	Core (Fault Isolation Register [FIR] Unit)	td_cp_fir	Core
x'04'	L2 Slice	v_cerrs	STS/BIU
x'08'	I/O	z_ei_scom	I/O
x'0A'	BIU Controller	t_gusras_reg	STS/BIU
x'40'	Power-On Reset (POR)	tc_por	ChipRAS
x'50'	Global Controls (Free Running)	ts_glob	ChipRAS
x'60'	SCOM Mode/Status (Free Running)	t_pscom_cntl	Always available
x'8[0:4]'	Clock Controls (Free Running)	tc_ccintf	Always available

Table 12-4 lists the modifier address decodes.

Table 12-4. SCOM Modifier Addresses (Page 1 of 3)

Modifier SCOM Address (9:16)	Register	Domain	See Page
x'021001'	CoreRAS Control (Pulsed) Register	Core	394
x'021100'	CoreRAS Mode Register	Core	396
x'021200'	CoreRAS Status Register	Core	400
x'021301'	Core Hang-Recovery Control Register	Core	403
x'021400'	Core Power Down and Idle Status Register	Core	406
x'022001'	Service Processor Special Attention (SP-ATTN) Register	Core	407
x'022100'	Service Processor And-Mask Register	Core	407
x'022200'	Service Processor Or-Mask Register	Core	407
x'022601'	Asynchronous Machine-Check Source Register	Core	408
x'022700'	Asynchronous And-Mask Register	Core	408
x'022800'	Asynchronous Or-Mask Register	Core	408
x'023000'	Instruction Address Breakpoint Register	Core	409
x'023101'	Hardware Implementation Dependent Register 0 (HID0)	Core	410
x'023201'	Hardware Implementation Dependent Register 1 (HID1)	Core	411
x'023300'	Instruction Match CAM (IMC) Register	Core	412
x'023401'	Patch Map (IMC Write Control Register)	Core	413
x'023500'	Hypervisor Decrementer	Core	414

Table 12-4. SCOM Modifier Addresses (Page 2 of 3)

Modifier SCOM Address (9:16)	Register	Domain	See Page
x'023600'	Time Base Register	Core	415
x'024001'	Performance Monitor Sampling Control Register	Core	416
x'030001'	Core Fault Isolation Register	Core	417
x'031000'	Core And-Mask Register	Core	417
x'032000'	Core Or-Mask Register	Core	417
x'030400'	Core Fault Isolation Mask Register	Core	421
x'031401'	Core And-Mask Register	Core	421
x'032401'	Core Or-Mask Register	Core	421
x'030800'	Core Checkstop Enable Registers	Core	422
x'030901'	Core Machine-Check Enable Register	Core	423
x'036001'	Instruction Mark Configuration Register	Core	424
x'040000'	L2 Fault Isolation Register	STS/BIU	426
x'041001'	L2 Fault Isolation And-Mask Register	STS/BIU	426
x'042001'	L2 Fault Isolation Or-Mask Register	STS/BIU	426
x'040801'	L2 Fault Isolation Checkstop Register	STS/BIU	426
x'040401'	L2 Error Mask Register	STS/BIU	426
x'041400'	L2 Error And-Mask Register	STS/BIU	426
x'042400'	L2 Error Or-Mask Register	STS/BIU	426
x'040801'	L2 Checkstop Enable	STS/BIU	426
x'043000'	BIU Mode Register	STS/BIU	434
x'083000'	PI Mode Register 0	I/O	436
x'083101'	PI Mode Register 1	I/O	437
x'083201'	PI Mode Register 2	I/O	438
x'083300'	PI Mode Register 3	I/O	439
x'084001'	PI Status Register	I/O	440
x'085000'	PI Command Register	I/O	444
x'086000'	Driver Initial Alignment Pattern (IAP) Register	I/O	445
x'086101'	Receiver IAP Register	I/O	445
x'0A0001'	BIU Fault Isolation Register/And-Mask/Or-Mask	STS/BIU	429
x'0A0400'	BIU Error Mask/And-Mask/Or-Mask	STS/BIU	431
x'0A0800'	BIU Checkstop Enable	STS/BIU	432
x'0A8000'	Processor Configurable Timing Delay Parameter Register (BUSCONF)	STS/BIU	373
x'0A9000'	BIU Status Register	STS/BIU	433
x'400000'	Power-On Reset Status Register	ChipRAS	446
x'400101'	Power-On Reset Continue Register	ChipRAS	448
x'400201'	Power-On Reset I^2C/JTAG Arbitration Register	ChipRAS	449

Table 12-4. SCOM Modifier Addresses (Page 3 of 3)

Modifier SCOM Address (9:16)	Register	Domain	See Page
x'400801'	Power-Management Control	ChipRAS	450
x'401400'	Power-On Reset Sequence Register 0	ChipRAS	452
x'402400'	Power-On Reset Sequence Register 1	ChipRAS	453
x'404400'	Power-On Reset Sequence Register 2	ChipRAS	454
x'408001'	Power Tuning Status Register	ChipRAS	455
x'500001'	Global Fault Isolation for Checkstop Conditions (Global FIR)	ChipRAS	456
x'500400'	Error Enable Mask	ChipRAS	457
x'500601'	Mode Register for Fault Isolation Registers	ChipRAS	458
x'500700'	Debug Mode Register	ChipRAS	459
x'503001'	Hang Pulse Generation	ChipRAS	461
x'503100'	Early Hang Pulse Generation	ChipRAS	462
x'504101'	Chip ID Register	ChipRAS	463
x'600001'	SCOM Mode Register	Always available	464
x'600100'	SCOM Controller Error Register	Always available	466
x'600200'	Electronic Chip ID	Always available	468
x'600400'	Clock Ratio Register (N:1 Phase Hold Control)	Always available	469
x'800000'	Clock Command Register	Always available	470
x'800003'	Status Register	Always available	472
x'800006'	Phase Synchronization Control Register	Always available	474
x'800009'	Clock Command Control Register	Always available	475
x'80000A'	Energy Star Register	Always available	480
x'80000C'	Status Register Mask	Always available	482
x'80000F'	I/O Control Register	Always available	483
x'820004'	ABIST Status Register	Always available	484
x'840002'	LBIST Options Register	Always available	485
x'840008'	LBIST Channel Length Register	Always available	487
x'84000B'	LBIST Test Length Register	Always available	488
x'84000D'	Clock Ramping Configuration Register	Always available	489

12.2.1 Register Description Conventions

The descriptions of the SCOM registers use the following terms:

Name Refers to the instantiated latch name that will show up in the SCAN_DEF. The facility name enclosed in brackets is the data-out pin that can be referenced in the all event trace (AET) waveform file format.

Reserved Indicates that the latch might be implemented. The customer should write zeros and expect unknown data.

Not Implemented Indicates that the latch is not implemented (N/I). The customer should write zeros and expect zeros.

Type SCOM request type:

- **RW** Read/Write
- **RO** Read Only. Write requests to a read-only SCOM register are treated as NOPs, and the data is thrown away. A good response is returned.
- **WO** Write Only. Read requests to a write-only SCOM register will result in an error condition.
- **RWor** Read/Write OR. The write operation is a special type that will OR into the specified bits in the register. An associated write-only AND mask is provided to enable clearing bits in this type of SCOM register.

12.2.2 SCOM Error Handling

If an error occurs while servicing an SCOM command, the ANY_SCATTN bit in the Access Status Register is raised. The service processor unit (SPU) should issue an SCOM reset by using the Instruction Register (IR) operation code x'1B'. Then, the SPU should read the SCOM Controller Error Register (x'600100') to see what the fault was. To clear ANY_SCATTN, first write all zeros to the SCOM Controller Error Register. Then write all zeros to the JTAG SCOM Status Register (x'000080'). Finally write all zeros to the JTAG Access Status Register (x'000002').

The following error indications in bits 0 through 23 of the SCOM Controller Error Register might be due to programming errors:

x'010000' Invalid address. Does not match any known address ranges.

x'001041' The address was decoded as a serial type and sent onto the SCOM serial ring, but no SCOM satellite accepted ownership of the address. This is probably due to the use of an invalid address.

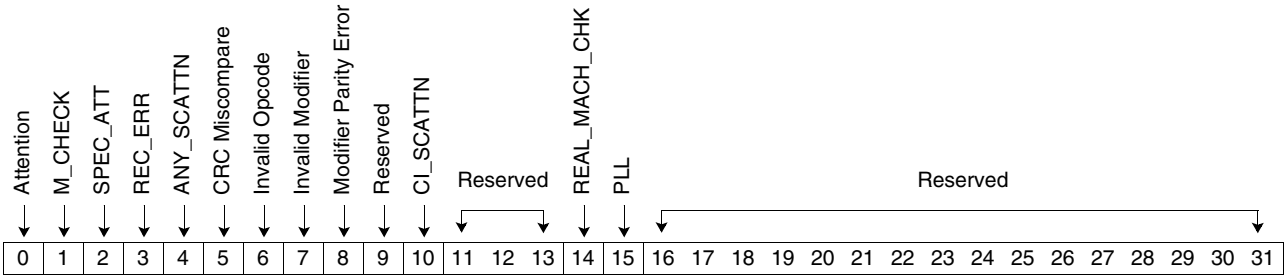
x'001040' The address was decoded as a serial type read, but no data was returned. This is probably due to an invalid read request being sent on the serial SCOM bus. This error, for example, will happen if the customer attempts to read a write-only register.

Bits 24 through 27 of the SCOM Controller Error Register contain the failing SCOM address.

12.2.3 Access Status Register

The Access Status Register contains bits that indicate error states that might occur during instruction or data scanning. This register is flushed to all zeros during POR.

Modifier Address x'000002'



Bits	Field Names	Description
0	Attention	(Master) The source for this bit is ATTENT_DR.
1	M_CHECK	Input to Access from the chip logic. **Note:** This is the system checkstop.
2	SPEC_ATT	SPEC_ATT input to Access from the chip logic.
3	REC_ERR	REC_ERR input to Access from the chip logic.
4	ANY_SCATTN	The OR of all SCOM attentions.
5	CRC Miscompare	If set, a scan data check miscompare was found on a scan-in operation.
6	Invalid Opcode	The IR opcode is not supported.
7	Invalid Modifier	The instruction does not support the received modifier.
8	Modifier Parity Error	Odd parity is required across bits 8 - 31 of the Instruction Register (the modifier address), except as noted in the Instruction Register description.
9	Reserved	Reserved.
10	CI_SCATTN	CI_SCATTN attention from the clock tree SCOM logic located in the ChipRAS clock control macro, ccintf.
11:13	Reserved	Reserved.
14	REAL_MACH_CHK	Machine check attention from the PowerPC 970 core.
15	PLL	PLL has lost lock.
16:31	Reserved	Reserved (not writable).

12.3 Core Pervasive SCOM Register Definitions

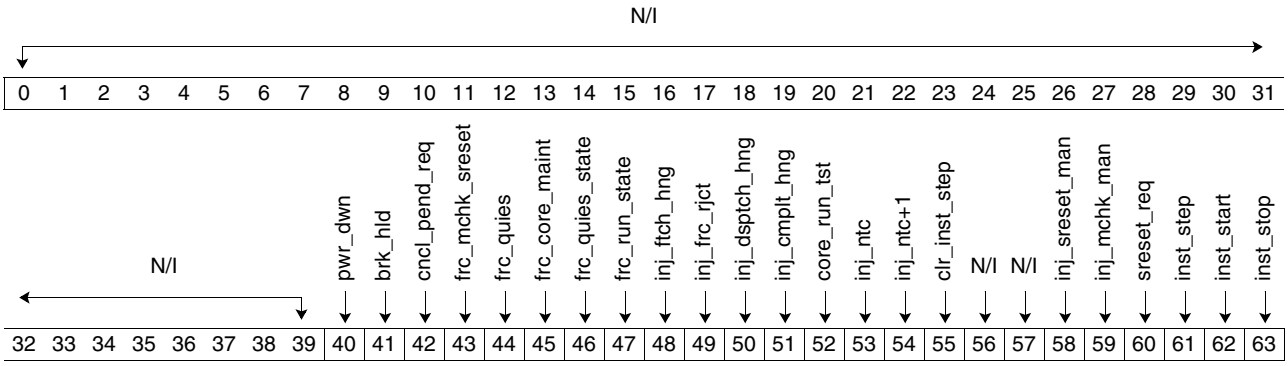
12.3.1 Processor CoreRAS Facilities (x'02[1:4]XXX')

CoreRAS Control (Pulsed) Register

Address x'021001'

Type WO

Reset N/A



Bits	Field Name	Description
0:39	N/I	Not implemented.
40	pwr_dwn	Enter core Power Down mode. This works without setting the Power Management bit of the MSR Register (MSR[POW]) (see bits 5:6 of the *CoreRAS Mode Register* on page 396).
41	brk_hld	Break hold on IFU/LSU status after core hang detect due to external source (see bits 59:63 of the *CoreRAS Status Register* on page 400).
42	cncl_pend_req	Cancel pending requests (quiesce, soft reset, machine check, core step). Go ahead and accept pending mode changes as well.
43	frc_mchk_sreset	Force the next machine check or soft reset (and all interrupts until then) to be marked nonrecoverable.
44	frc_quies	Force the core to quiesce manually (RAS LOGIC OVERRIDE).
45	frc_core_maint	Force core maintenance mode (RAS LOGIC OVERRIDE).
46	frc_quies_state	Force the quiesce state machine to the Quiesce state (RAS LOGIC OVERRIDE).
47	frc_run_state	Force the quiesce state machine to the Run state (RAS LOGIC OVERRIDE).
48	inj_ftch_hng	Inject fetch hang to test hang-recovery logic.
49	inj_frc_rjct	Inject force reject hang to test hang-recovery logic.
50	inj_dsptch_hng	Inject dispatch hang to test hang-recovery logic.
51	inj_cmplt_hng	Inject completion hang to test hang-recovery logic.
52	core_run_tst	Core running test. Use to see if the core is running (clears bit 15 of the CoreRAS Status Register until a group completes). **Note:** Do not do this if in Maintenance Single Step mode (bit 54 of the CoreRAS Status Register must equal '0').

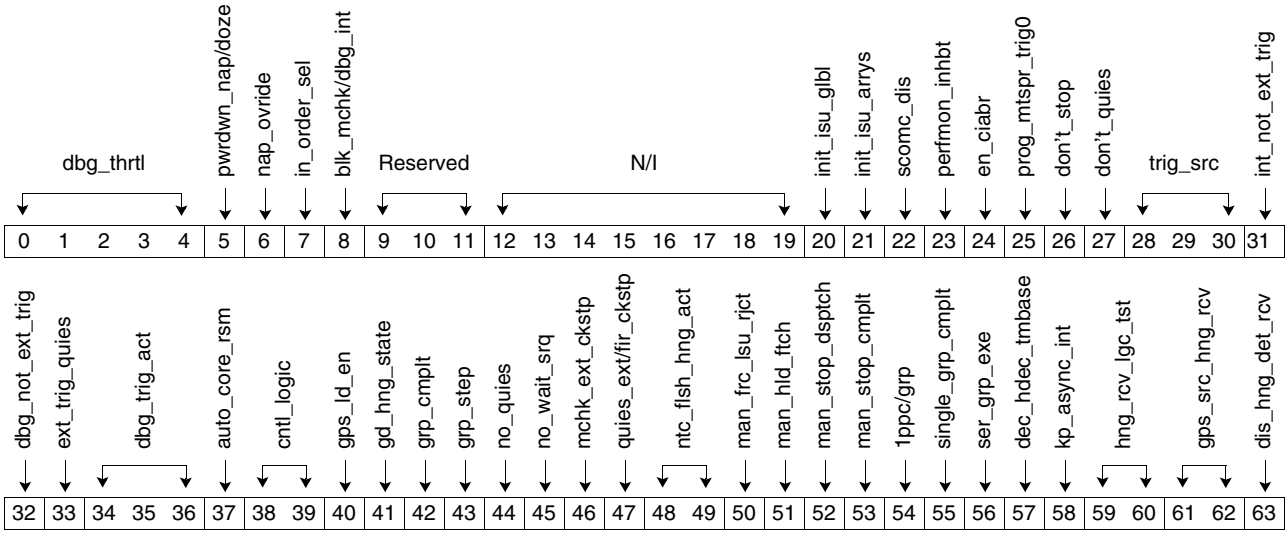
Bits	Field Name	Description
53	inj_ntc	Inject next to complete (NTC) for all instructions in the machine. Flush manually without the before and after waits. **Note:** Do not attempt if the processor is running.
54	inj_ntc+1	Inject next to complete plus one (NTC + 1) in all but the oldest instruction. Flush manually without the before and after waits. **Note:** Do not attempt if the processor is running.
55	clr_inst_step	Clear instruction stop due to checkstop. Also clears hang detection, hang history, and miscellaneous status latches.
56	N/I	Not implemented.
57	N/I	Not implemented.
58	inj_sreset_man	Inject *sreset* manually (no auto quiesce). The ISU can OR x'0100' with another interrupt vector if they occur simultaneously. **Note:** Do not attempt if the processor is running.
59	inj_mchk_man	Inject machine check manually (no auto quiesce). The ISU can OR x'0200' with another interrupt vector if they occur simultaneously. **Note:** Do not attempt if the processor is running.
Bit 15 of the CoreRAS Status Register will be cleared for bits 60:62 until a group completes (that is, until positive acknowledgment is received that the operation was successful).		
60	sreset_req	SRESET request. This causes the core to first quiesce, and then vector to x'0100' and start instructions. If quiesce is unsuccessful, a soft reset will not occur. A special attention will be sent to the service processor indicating a timeout on a quiesce request (assuming hang pulses are activated). After the request, check that bit 15 of the CoreRAS Status Register equals '1' to ensure that the soft reset was successful (core has started).
61	inst_step	Instruction step (core step). **Notes:** • The core must be in Maintenance mode (quiesced) (see bit 12 of the *CoreRAS Status Register* on page 400). • Single Group Completion mode is active, which means a core flush and refetch occurs between each step. This allows the next instruction address (NIA) to be changed (via scan) between steps if wanted. • Bit 43 of the CoreRAS Register determines the behavior. It is *not* determined by bit 54 of the CoreRAS Mode Register or bit 0 of Hardware Implementation Dependent Register 0 (HID0). 0 PowerPC instruction step (default). Completes more than one group if the PowerPC instruction is a microcoded, multi-group sequence. 1 Group step (for debug). Completes a single group or microcode group sequence. After the request, check that bit 15 of the CoreRAS Status Register equals '1' to ensure that step was successful (core has started).
62	inst_start	Instruction start (core resume). **Notes:** • Core must be in Maintenance mode (quiesced) (see bit 12 of the *CoreRAS Status Register* on page 400). • After the request, check that bit 15 of the CoreRAS Status Register equals '1' to ensure that start was successful (core has started).
63	inst_stop	Instruction stop (core stop). **Note:** Causes core quiesce, and leaves core in Maintenance mode.

CoreRAS Mode Register

Address x'021100'

Type RW

Reset Reset to all zeros during POR.



Bits	Field Name	Description
0:4	dbg_thrtl	Debug throttle modes. Periodically alter instruction flow based on Core Debug Throttle Control. 00000 None. 01xxx Stop fetch (both IFU and LSU prefetch). 0x1xx Stop dispatch. 0xx1x Stop completion. 0xxx1 Force LSU reject (stops issue from ISU to LSU). 11xxx One PowerPC per group (can quiesce depending on bit 44). 1x1xx Single group completion (will quiesce to change). 1xx1x Serialized group issue (will quiesce to change). 1xxx1 Serialized group dispatch (does *not* quiesce to change).
5	pwrdwn_nap/doze	Power down Nap/Doze mode enable override. (When set, ignores bits 8:9 of the Hardware Implementation Dependent Register 0 [HID0]) (see bit 6 for mode select). **Note:** To enter Power Down, software must either set MSR[POW] or use CoreRAS Control Register[40].
6	nap_ovride	Nap mode override selector. 0 Force Nap mode when bit 5 is set. 1 Force Doze mode when bit 5 is set.
7	in_order_sel	In-order issue select. 0 Serialized Group Execution, when selected, will serialize at dispatch. This is the most powerful form of serialization, and only allows one group in flight in the machine at a time. This includes branch and Condition Register (CR)-logical instructions. It also prevents groups from sharing ISU resources such as mappers and renames. 1 Serialized Group Execution, when selected, will serialize at issue. This does not include branches and CR-logical instructions and does not completely serialize the ISU.
8	blk_mchk/dbg_int	Blocks machine-check interrupt or debug interrupt injection based on debug triggers when one is already set in the Asynchronous Machine-Check Source Accumulation Register. This will prevent multiple triggers from causing a checkstop with machine-check enable (ME) equal to '0'. It allows the interrupt handler to ignore spurious triggers until it has a chance to clear the source register.

Bits	Field Name	Description
9:11	Reserved	Spare.
12:19	N/I	Not implemented.
20	init_isu_glbl	Initialize ISU global completion table.
21	init_isu_arrys	Initialize ISU SBM, SBX, and SFM arrays.
22	scomc_dis	SCOMC disable. This is a debug switch used to disable the core SCOM master accessible through the core SPR bus. RAS logic treats an MTSCOMC instruction in the core as a NOP.
23	perfmon_inhbt	Performance monitor inhibit (debug switch only; interrupts and multiplexer selects).
24	en_ciabr	Enable CIABR (bit 22 in Hardware Implementation Dependent Register 0 [HID0] override active).
25	prog_mtspr_trig0	Programmable MTSPR TRIG0 mode. 0 SPR 976 mtspr_data is *not* used to form TRIG1 and TRIG2. 1 SPR 976 mtspr_data(63) causes TRIG1, and mtspr_data(62) causes TRIG2. **Note:** All zeros in SPR 976 mtspr_data(0:63) always cause TRIG0.
26	don't_stop	Do not stop fetch, dispatch, and completion on checkstop. Set this bit when the chip is set to clock stop on a checkstop. Setting this bit prevents core scan rings from being corrupted when a checkstop occurs, because it takes several cycles to clock stop after a checkstop occurs.
27	don't_quies	Do not attempt to quiesce, and then machine check or soft reset, during core hang recovery. **Note:** This will cause a checkstop if the first two attempts are unsuccessful.
RAS trigger sources to debug logic		
28:30	trig_src	000 No internal trigger selected. 001 External trigger causes internal trigger. 010 Decrementer interrupt causes internal trigger. 011 External interrupt causes internal trigger. 100 Machine-check interrupt causes internal trigger. 101 Problem-state hang detect causes internal trigger. 110 Bad-state hang detect causes internal trigger. 111 Quiesce causes internal trigger.
31	int_not_ext_trig	Internal trigger does *not* cause an external trigger.
32	dbg_not_ext_trig	Core DBG trigger does *not* cause an external trigger.
RAS activities based on debug trigger		
33	ext_trig_quies	External trigger causes a core quiesce. Bit 37 determines behavior after the quiesce.
34:36	dbg_trig_act	Core DBG trigger causes: 000 No action selected. 001 Core quiesce (bit 37 determines behavior after the quiesce). 010 NTC flush (see bits 46:48 of the *Core Hang-Recovery Control Register* on page 403). 011 NTC + 1 flush (see bits 46:48 of the *Core Hang-Recovery Control Register* on page 403). 100 Machine-check interrupt (see bit 8 for mode to prevent checkstop). 101 Debug interrupt. 110 Enter Reduced Execution mode for a specified number of group completions (see bits 46:48 of the *Core Hang-Recovery Control Register* on page 403). 111 Pause (quiesce and hold prefetch) until core idle.
37	auto_core_rsm	Auto core resume on core quiesce due to debug trigger. When debug or an external trigger causes quiesce: 0 Quiesce, set Special ATTN, and enter Core Maintenance mode. 1 Auto-start core immediately after ISU completes the quiesce, which implicitly causes a flush. Special ATTN is not set, and the core resumes normal instruction execution.

Bits	Field Name	Description
RAS control logic modes		
38:39	cntl_logic	00 No hang-pulse induced random periods. 10 Enter Reduced Execution mode for a random period after every hang pulse (see bits 46:48 of the *Core Hang-Recovery Control Register* on page 403). 01 Quiesce for a random period after every hang pulse. 11 Undefined.
40	gps_ld_en	STS load mode enable. Throttle back core execution when the STS detects a memory subsystem hang.
41	gd_hng_state	Return to the Good Hang state immediately after meeting the completion criteria. Do *not* wait on hang pulse to leave Reduced Throughput mode.
42	grp_cmplt	Use any group completed (instead of PowerPC) to debug logic (trace trigger) and to indicate progress for the hang detection logic. 0 PowerPC group completed. 1 Any group completed (including intermediate multi-group sequences).
43	grp_step	Group step instead of PowerPC instruction step. Causes the instruction step *not* to enter a PowerPC mode automatically during Maintenance mode.
44	no_quies	Do not quiesce when changing a PowerPC mode.
45	no_wait_srq	Do not wait for the store reorder queue (SRQ) to be empty before auto-restarting after a quiesce due to a mode change, debug trigger, or when in single_group_completion mode.
46	mchk_ext_ckstp	Machine check on an external checkstop, required for logical partitioning (LPAR).
47	quies_ext/fir_ckstp	Quiesce on external or FIR checkstop (default is hold completion). **Note:** Causes Special ATTN to the service processor.
48:49	ntc_flsh_hng_act	Action to perform for NTC flush hang recovery when the LSU is not safe (that is, the load miss queue [LMQ] is not empty). 00 Do not attempt flush. This action continues to hold completion, dispatch, and force reject at the same time. It eventually transitions to the next Hang-Recovery state if the LSU is not safe. 01 Stop dispatch and completion (sets the Service Processor Special Attention (SP-ATTN) Register). 10 Checkstop. 11 Do flush anyway. **Caution:** For debug only. Use the Unsafe NTC + 1 Mode (bit 17 of the Core Hang-Recovery Control Register) for NTC + 1 flushes.
Core toolbox -- hooks for the service processor		
50	man_frc_lsu_rjct	Manually force LSU reject.
51	man_hld_ftch	Manually hold fetch (both IFU and LSU).
52	man_stop_dsptch	Manually stop dispatch.
53	man_stop_cmplt	Manually stop completion.
Core toolbox -- behavior mode controls **Note:** The core *must* be quiesced before changing these fields. A corresponding HID0 change invokes the state machine to perform an automatic quiesce, mode change, and resume.		
54	1ppc/grp	One PowerPC per Group mode. When set, ORed with HID0[0]. This means that no more than one PowerPC instruction will be placed in a group. Some instructions are expanded into a multiple-group, microcoded sequence.
55	single_grp_cmplt	Single Group Completion mode. When set, ORed with HID0[1], which causes an automatic group step. This means that only one group (or microcoded group sequence) is allowed to complete at a time. A core quiesce (involving an instruction flush and refetch) occurs between the completion of each group. Subsequent groups of instructions will not be allowed to complete, but *will* be allowed to execute at the same time unless mode (bit 56) equals '1'.

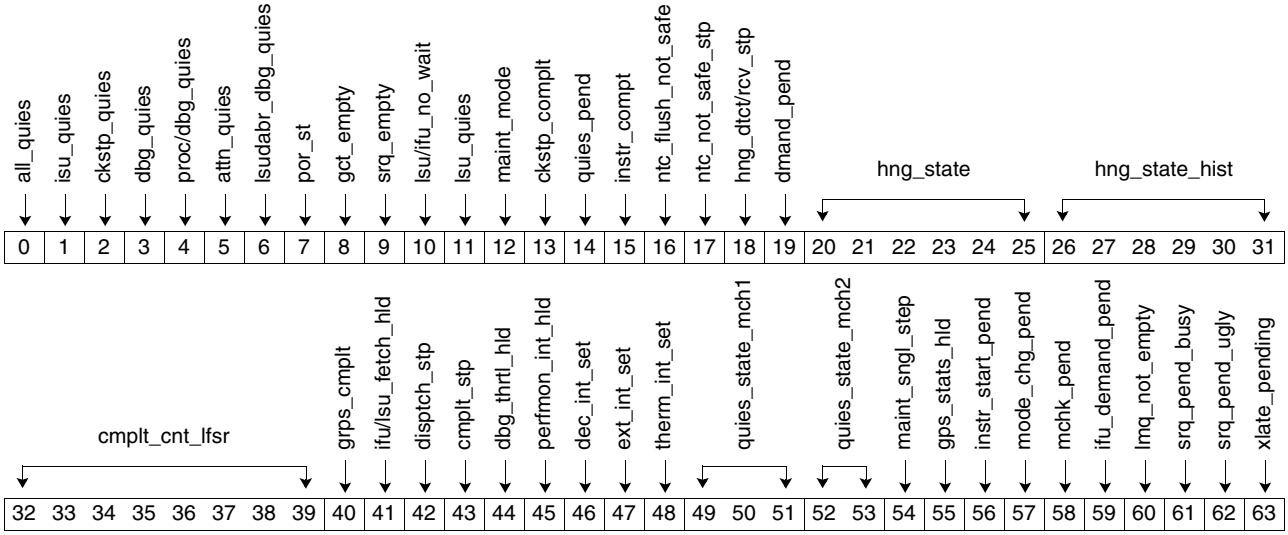
Bits	Field Name	Description
56	ser_grp_exe	Serialized Group Execution mode (in-order dispatch OR issue of groups). • ORed with HID0[3] when inorder_issue_select equals '0'. • ORed with HID0[16] when inorder_issue_select equals '1'. When set, this means that a group must be completed before the next one can be issued or dispatched, effectively serializing execution of groups in the processor. Subsequent groups of instructions are not allowed to execute at the same time (see inorder_issue_select, bit 7 of this register, for more details).
Core Maintenance Modes		
57	dec_hdec_tmbase	Run decrementer (DEC), hypervisor decrementer (HDEC), and time base while instruction stepping (take ISU quiesced into account during maint_mode). This also allows DEC and HDEC interrupts, and only these interrupts, while instruction stepping. **Note:** DEC, HDEC, and time-base stopping only takes effect if bit 18 of HID0 equals '0'.
58	kp_async_int	Keep asynchronous interrupts during quiesce or instruction stepping. Asynchronous interrupts include decrementer, external, external machine-check, and performance monitor interrupts. If kept, the interrupts present only after maintenance activity completes.
59:60	hng_rcv_lgc_tst	Hang-Recovery Logic Test modes. 00 None 01 Allow Problem Hang Recovery to break test hang. 10 Allow Bad Hang Recovery to break test hang. 11 Allow Machine-Check Hang Recovery to break test hang.
61:62	gps_src_hng_rcv	STS Source Core Hang-Recovery modes. 00 None (only attempt hang recovery if nothing is pending to the STS). 01 Special ATTN if ambiguous source or if either IFU or LSU indicates STS pending transactions. 10 Attempt core hang recovery if ambiguous source; there are possibly STS pending transactions. 11 Always attempt a core hang recovery, even if ambiguous source or either IFU or LSU indicates STS pending transactions. **Note:** To cause Special ATTN for core-source hang detect (no STS pending), use bit 55 of the Core Hang-Recovery Control Register.
63	dis_hng_det_rcv	Disable hang detection and recovery based on hang pulse.

CoreRAS Status Register

Address x'021200'

Type RO

Reset All zeros except bits 7, 12, 20, and 26.



Bits	Field Name	Description
0	all_quies	Entire core is quiesced (that is, status[1] is equivalent to an AND of status[8:10]).
1	isu_quies	The ISU is quiesced for the reasons listed below.
2	ckstp_quies	A checkstop caused the quiesce request.
3	dbg_quies	Debug logic made the quiesce request to the ISU.
4	proc/dbg_quies	The ISU completed quiesce as requested by the service processor or debug logic.
5	attn_quies	An ATTN instruction quiesced the ISU.
6	lsudabr_dbg_quies	The LSU DABR Debug mode quiesced the ISU. **Note:** CIABR will *not* set this bit, but bit 2 of the Service Processor Special Attention (SP-ATTN) Register will.
7	por_st	POR state (scan flush indicator). Forces stop dispatch. When set, indicates a quiesce *must* be performed before starting the core.
8	gct_empty	The ISU global completion table (GCT) is empty (quiesce indication).
9	srq_empty	The ISU SRQ is empty (quiesce indication).
10	lsu/ifu_no_wait	The LSU and IFU are *not* waiting for an STS transaction (NOR of the *not-held* but masked status sourcing bits [59:63]).
11	lsu_quies	The LSU is quiesced. • The LMQ is idle (there are no outstanding load misses, including speculative loads). • The SRQ has no stores past completion. The service processor should poll this bit to verify that both the STS and core are quiesced.

1. "Ugly" operations are unsafe instructions that are in flight, so that the machine state is not clean.

Bits	Field Name	Description
12	maint_mode	The core is in Maintenance mode; the ISU is quiesced or is being stepped. • Enter and stay in Maintenance mode when ISU quiesces (including during instruction stepping). • Leave Maintenance mode when core resume, soft reset, or external machine check occurs (see bit 59 of the *CoreRAS Mode Register* on page 396).
13	ckstp_complt	A checkstop stops completion.
14	quies_pend	A quiesce request is pending.
15	instr_compt	An instruction (or group) completed since the last maintenance operation (that is, an instruction step, start, or soft reset). For a core running test, first use bit 55 of the CoreRAS Control (Pulsed) Register [x'021001']).
16	ntc_flush_not_safe	An NTC flush is not safe and caused a checkstop (core HANG).
17	ntc_not_safe_stp	An NTC is not safe and caused a Stop (ATTN).
18	hng_dtct/rcv_stp	A hang detect/recovery caused a Stop (ATTN) on a Problem, Bad, or STS hang detect as programmed.
19	dmand_pend	A Demand Instruction fetch has been pending since the last hang pulse.
20:25	hng_state	Hang state Bit State 20 Good 21 Problem 22 Bad 23 Quiesce/Machine Check 24 Wait 25 Fail
26:31	hng_state_hist	Hang state history Bit State 26 Good 27 Problem 28 Bad 29 Quiesce/Machine Check 30 Wait 31 Fail
32:39	cmplt_cnt_lfsr	An 8-bit completion count LFSR. Matched against the Completion Count Limit in the Core Hang-Recovery Control Register (see page 403).
40	grps_cmplt	Set to one, once the programmed number of groups complete after Hang state. This bit clears on return to the Good state.
41	ifu/lsu_fetch_hld	The CoreRAS logic is holding Fetch on both the IFU and LSU.
42	disptch_stp	The CoreRAS logic has stopped Dispatch.
43	cmplt_stp	The CoreRAS logic has stopped Completion.
44	dbg_thrtl_hld	Work (instruction completion) is being held due debug throttle hooks to CoreRAS.
45	perfmon_int_hld	Performance monitor interrupt is being held pending completion of a core maintenance operation.
46	dec_int_set	Decrementer interrupt is set.
47	ext_int_set	External interrupt is set.
48	therm_int_set	Thermal interrupt is set.
49:51	quies_state_mch1	Quiesce state machine Bit State 49 Quiesced 50 Running 51 Asynchronous interrupt

1. "Ugly" operations are unsafe instructions that are in flight, so that the machine state is not clean.

Bits	Field Name	Description
52:53	quies_state_mch2	Quiesce state machine Bit State 52 Change mode 53 Pause
54	maint_sngl_step	Maintenance Single Stepping mode is in effect. The core is in Serial Dispatch mode (and automatically in a PowerPC mode if bit 43 of the CoreRAS Mode Register equals '0').
55	gps_stats_hld	The STS status (bits 59:63 of this register) is being held due to a core-STS hang detect for debug/fault isolation.
56	instr_start_pend	Instruction start (core resume) pending, or an instruction (or group) step pending.
57	mode_chg_pend	Mode change pending.
58	mchk_pend	Machine check pending.
Note: Bits 59 and 60:63 are maskable by scan latches (TDR.IFU_STAT_DIS and TDR.LSU_STAT_DIS). Normally, they follow the masked IFU/LSU signals. However, they are held when a core-STS hang detect occurs to aid in debug/fault isolation (indicated by bit 55). Use bit 41 of the CoreRAS Control (Pulsed) Register to break the hold after an STS hang detect occurs. The hold is also broken during hang recovery as soon as a single group completes. However, it can take more than one group complete to deem hang recovery successful. In this case, if the processor is still hung, these status indicators follow the current status.		
59	ifu_demand_pend	The IFU has outstanding demand instruction fetches to the STS.
60	lmq_not_empty	The LSU has outstanding loads to the STS (this also indicates that it is *not* safe to NTC flush).
61	srq_pend_busy	The LSU has entries in the SRQ being held due to an STS busy on a store port.
62	srq_pend_ugly	The LSU has outstanding ugly[1] operations to the STS waiting for response. That is, an instruction cache block invalidate (ICBI), store word conditional (STWCX), or SYNC instruction is outstanding to the STS.
63	xlate_pending	The LSU has an outstanding instruction or data-side table walk to the STS.

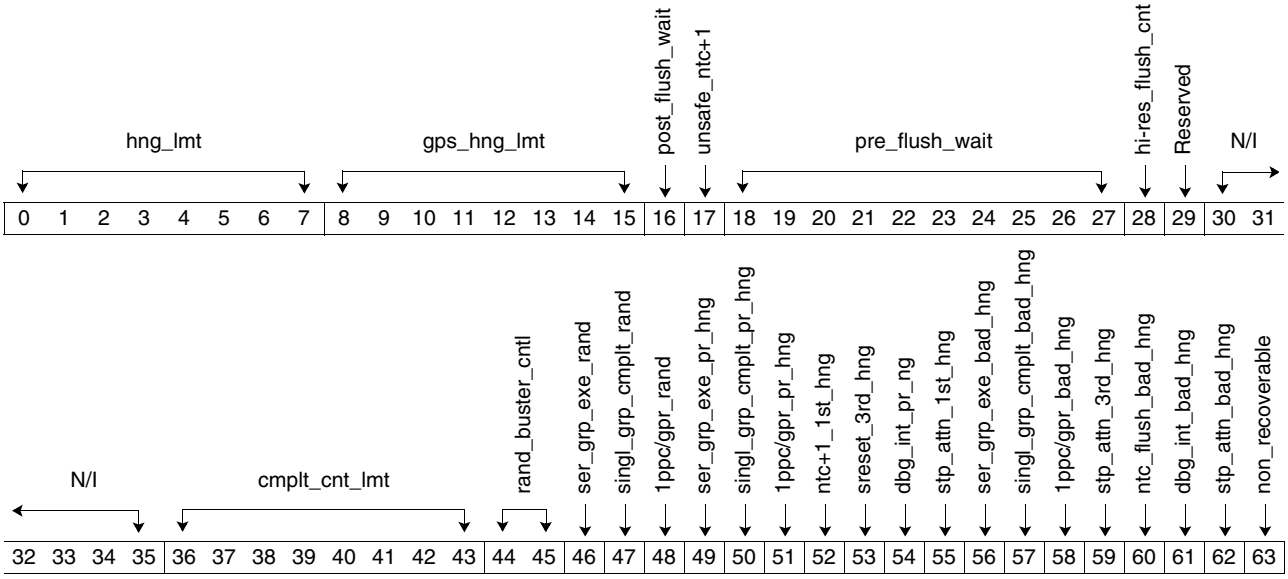
1. "Ugly" operations are unsafe instructions that are in flight, so that the machine state is not clean.

Core Hang-Recovery Control Register

Address x'021301'

Type RW

Reset
0:7 Reset to x'9F' = 3 hang pulses
8:15 Reset to x'2D' = 100 hang pulses
16:17 Reset to 00
18:27 Reset to x'1FF' = 1 × 255 default
36:43 Reset to x'FE' = 255 (maximum)
44:63 Reset to all zeros during POR, except bits 52 and 60.



Bits	Field Name	Description
0:7	hng_lmt	Core hang limit (8-bit LFSR match value). This is the number of hang pulses without a PowerPC instruction (or group) completion used to detect a core hang. It is used when the *lsu_safe* signal indicates there are no outstanding load/store (LD/ST) instructions to the STS. This should be set to a minimum of 3 (LFSR code value for 3 is x'9F').
8:15	gps_hng_lmt	Core-STS hang limit (8-bit LFSR match value). This is the number of hang pulses without a PowerPC instruction (or group) completion used to detect a core-STS hang. It is used when lsu_safe indicates outstanding load/store (LD/ST) instructions to the STS. This must be set to a value greater than the core hang limit indicated in bits 0:7 of this register.
16	post_flush_wait	After flush wait time. After flush control. 0 Wait 255 cycles after attempting hang-recovery flush. 1 Use the before flush wait time, defined in bits 18:27, as the after flush wait time as well.
17	unsafe_ntc+1	Unsafe NTC + 1 mode. 0 Do not attempt NTC + 1 flushes if not safe. 1 Attempt NTC + 1 even if the LSU indicates not safe (the LMQ is not empty).

Bits	Field Name	Description
18:27	pre_flush_wait	Before flush wait time. Wait time (x × 255 cycles) before hang-recovery flush (10-bit LFSR match value). Dispatch and completion (and LSU force reject for NTC) are stopped during this wait *before* attempting flush. 1FF 1 × 255 = 255 cycles (minimum wait). 3FE 1022 × 255 = 260610 cycles (maximum wait). **Note:** 3FF (count of zero) is invalid and will cause undefined results.
28	hi-res_flush_cnt	High-resolution flush counter (before and after wait times are single cycle instead of having a 255 multiplier).
29	Reserved	Spare.
30:35	N/I	Not implemented.
36:43	cmplt_cnt_lmt	Completion count limit (8-bit LFSR match value). This is the number of group completions to wait before returning to Good Hang state after a hang recovery is attempted.
Random pulse hang buster controls		
44:45	rand_buster_cntl	Random buster controls (use before and after wait times specified above). 00 Random pulse has no effect (disabled). 01 Random pulse injects NTC + 1 flush. 10 Random pulse injects NTC flush. **Caution:** A random NTC flush does not work in all cases. It should only be attempted in the bring-up lab. 11 Random pulse stalls instruction dispatch and completion with no flush (only waits for the before flush time). **Note:** Bit 17 of this register and bits 48:49 of the CoreRAS Mode Register determine the behavior of these flushes when LSU indicates not safe. The recommended setting is to not flush when LSU is not safe if any of these Random Busters are enabled.
Degraded mode during random period (either random pulse or hang pulse induced) **Notes:** • Bit 38 of the CoreRAS Mode Register must be set in order to enter these random periods. • These bits are also set until the programmed number of completions (above) have occurred after the flush on trigger. • inorder_issue_select (bit 7 of the CoreRAS Mode Register) affects the meaning of bits 46 and 49.		
46	ser_grp_exe_rand	Serialized Group Execution mode during random period (or after flush on trigger).
47	singl_grp_cmplt_rand	Single Group Completion mode during random period (or after flush on trigger).
48	1ppc/gpr_rand	One PowerPC per Group mode during random period (or after flush on trigger).
Degraded mode during first-level (problem) hang recovery (hang pulse based only)		
49	ser_grp_exe_pr_hng	Serialized Group Execution mode during problem hang recovery.
50	singl_grp_cmplt_pr_hng	Single Group Completion mode during problem hang recovery.
51	1ppc/gpr_pr_hng	One PowerPC per Group mode during problem hang recovery.
First-level (problem) hang state recovery actions (hang pulse based only)		
52	ntc+1_1st_hng	Attempt NTC + 1 flush for first hang detect (Problem Hang state). Default to '1'.
53	sreset_3rd_hng	Perform SRESET instead of machine check (MCHK) for third-level hang recovery, unless disabled by bit 27 of the CoreRAS Mode Register.
54	dbg_int_pr_ng	Cause maintenance (debug) interrupt to be taken after problem hang recovery.
55	stp_attn_1st_hng	Stop completion and cause special attention on first-level hang detect (sets the Service Processor Special Attention [SP-ATTN] Register).

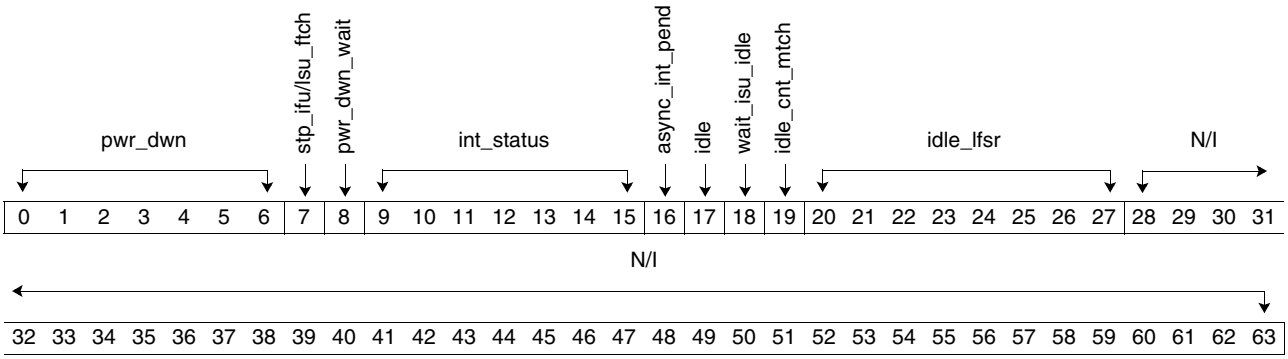
Bits	Field Name	Description
Degraded mode during second-level (bad) hang recovery		
56	ser_grp_exe_ bad_hng	Serialized Group Execution mode during bad hang recovery.
57	singl_grp_cmplt_ bad_hng	Single Group Completion mode during bad hang recovery.
58	1ppc/gpr_bad_hng	One PowerPC per Group mode during bad hang recovery.
Second-level (bad) Hang state recovery actions		
59	stp_attn_3rd_hng	Stop completion and cause special attention on third-level hang (sets the Service Processor Special Attention (SP-ATTN) Register). **Note:** Should also set bit 27 of the CoreRAS Mode Register.
60	ntc_flush_bad_hng	Attempt NTC flush on a bad hang. Defaults to '1'.
61	dbg_int_bad_hng	Causes a maintenance (debug) interrupt to be taken after a bad hang recovery.
62	stp_attn_bad_hng	Stop completion and cause a special attention on a bad hang (sets the Service Processor Special Attention (SP-ATTN) Register).
Nonrecoverable machine check for hang-recovery control		
63	non_recoverable	Make all hang-recovery MCHKs or SRESETs nonrecoverable (only has an effect if the quiesce was successful).

Core Power Down and Idle Status Register

Address x'021400'

Type RO

Reset Bits 0, 7, and 17 are initialized to '1'.



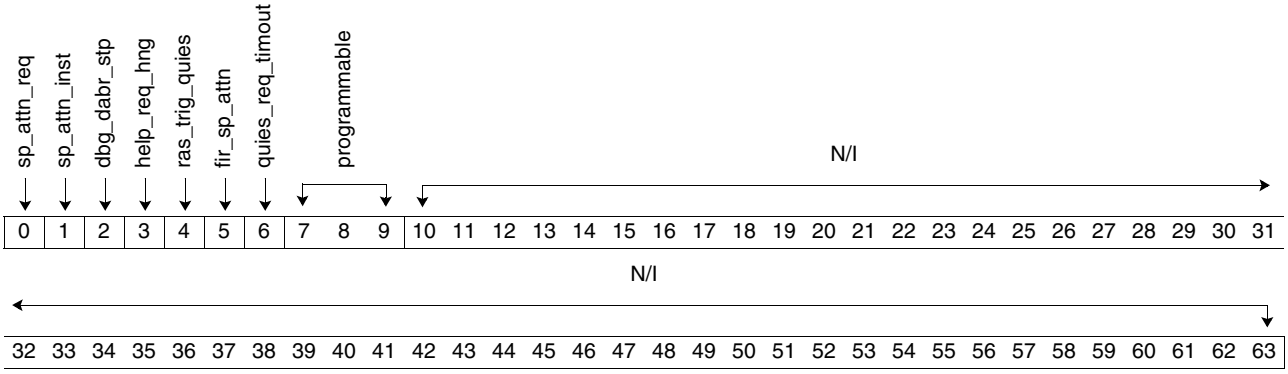
Bits	Field Name	Description
0:6	pwr_dwn	Power Down State Machine (0:6).
7	stp_ifu/lsu_ftch	IFU fetch and LSU prefetch stopped.
8	pwr_dwn_wait	Power down wait timer.
9:15	int_status	Interrupt status. Bit 9 Thermal interrupt active. 10 External interrupt active. 11 External machine check active or held, or machine check pending. 12 External *sreset* active or held, or *sreset* pending. 13 Performance monitor interrupt held. 14 Decrementer interrupt active. 15 Reserved.
16	async_int_pend	Asynchronous interrupt pending in the ISU.
17	idle	Core idle indication.
18	wait_isu_idle	Wait for idle from ISU.
19	idle_cnt_mtch	Idle count match.
20:27	idle_lfsr	Idle LFSR value.
28:63	N/I	Not implemented.

Service Processor Special Attention (SP-ATTN) Register
Service Processor And-Mask Register
Service Processor Or-Mask Register

Any bit set in this register will drive a special attention to the service processor. The processor core can use this register to implement a mailbox function to communicate with the service processor.

Note: This is an accumulation register, and each source must be cleared to deassert the special attention. The AND mask should be used to clear bits in the register. The OR-mask should be used by software to set new bits in order to prevent an attention from another source from being missed.

Address	x'022001' x'022100' (AND) x'022200' (OR)
Type	RW WO (AND) WO (OR)
Reset	Reset to all zeros during POR.



Bits	Field Name	Description
0	sp_attn_req	Core-to-service-processor special-attention request.
1	sp_attn_inst	SP-ATTN instruction. The ISU is quiesced for maintenance due to a software breakpoint.
2	dbg_dabr_stp	A debug DABR soft stop (or ISU CIABR) caused the core quiesce.
3	help_req_hng	Core hang detected results from a CoreRAS help request to the service processor (see *CoreRAS Status Register* on page 400). Bit 41 of the CoreRAS Mode Register and bits 55 and 62 of the Core Hang-Recovery Control Register determine the mode. As a precaution, core dispatch and completion are stopped while this bit is set.
4	ras_trig_quies	RAS trigger caused core quiesce.
5	fir_sp_attn	Core FIR-induced Special ATTN.
6	quies_req_timout	Time out on quiesce request (or pending change needing a quiesce). Two hang pulses have occurred and the request is still pending.
7:9	programmable	Programmable by the software or operating system.
10:63	N/I	Not implemented.

Asynchronous Machine-Check Source Register
Asynchronous And-Mask Register
Asynchronous Or-Mask Register

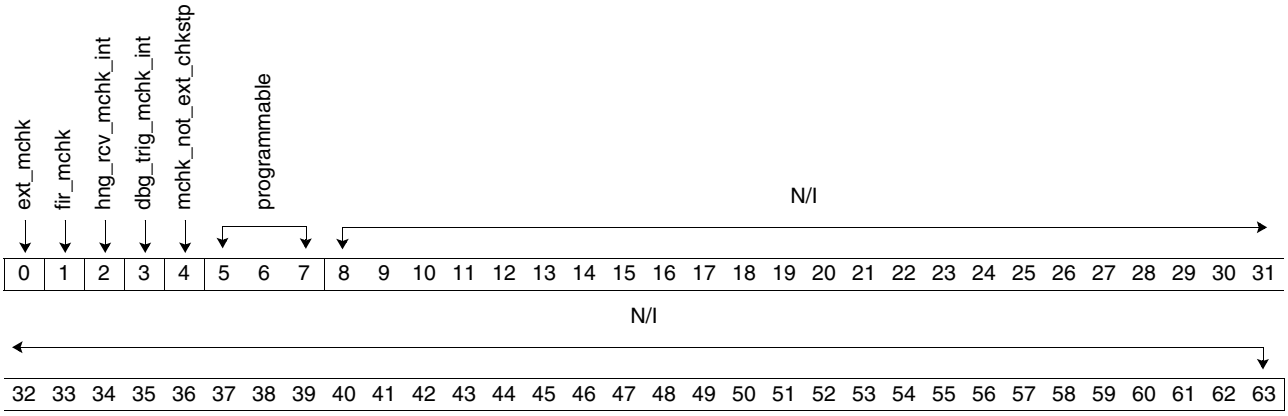
This register indicates the source of all asynchronous machine-check events. When any source event occurs, quiesce, then vector to x'0200', and restart instruction execution. If the quiesce is unsuccessful, a machine check will *not* occur. A special attention will be sent to the service processor indicating a timeout on the quiesce request (assuming hang pulses are activated). The service processor can then check bit 58 of the CoreRAS Status Register to see if the machine check was accepted or is still pending.

Note: This is a history accumulation register and must be cleared after each interrupt to absolutely determine the new source. The AND mask should be used to clear bits in the register. Software should use the OR mask to set new bits to prevent an interrupt from another source from being missed.

Address x'022601'
x'022700' (AND)
x'022800'(OR)

Type RW
WO (AND)
WO (OR)

Reset Reset to all zeros during POR.



Bits	Field Name	Description
0	ext_mchk	External machine check from the chip C4 pin.
1	fir_mchk	FIR induced machine check (see *Section 12.3.4 Processor Core FIR Facilities (x'03[0:5]XXX')* on page 417).
2	hng_rcv_mchk_int	Hang-recovery machine-check interrupt attempt.
3	dbg_trig_mchk_int	Debug-logic trigger machine-check interrupt.
4	mchk_not_ext_ chkstp	Machine check instead of external checkstop (see the *CoreRAS Mode Register* on page 396). Intended for the LPAR.
5:7	programmable	Programmable by software or the operating system.
8:63	N/I	Not implemented

12.3.2 Processor Core SPR SCOM Access (x'023XXX')

Instruction Address Breakpoint Register

The Instruction Address Breakpoint Register (IABR) supports the address-breakpoint instruction. Writing this SCOM register *only* sets latches in the IFU, which can also be set by scan (GCP.PIFU.IFBT.IABR.L2[0:63]).

An IABR match occurs on the fetch of any instruction, even a speculative instruction. By default, the IABR matches if the address in the current fetch group is equal to or *after* the current IFAR. There can be multiple IABR matches for a single instruction before it is actually executed (or completed).

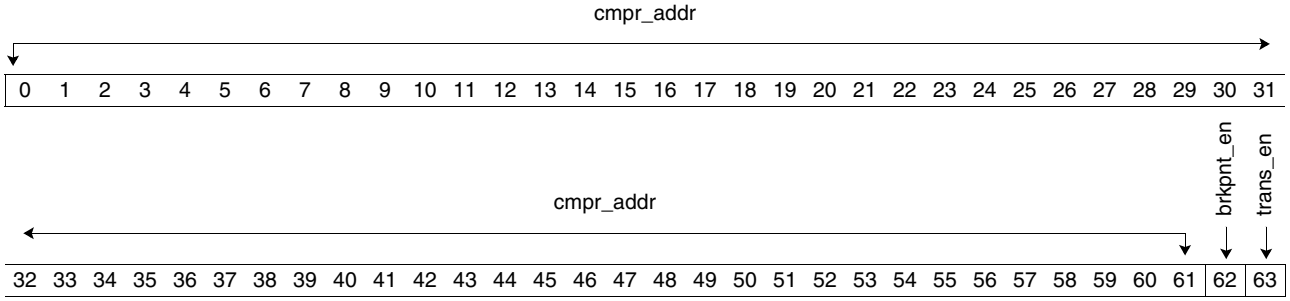
An additional mode bit accessible through scan only forces exact matches (GCP.PIFU.IFBC.ASSSCANONLYNEW[7]).

Note: This register uses the IFU FETCH address, not the current instruction address (CIA) that is executing.

Address x'023000'

Type WO

Reset Reset to all zeros during POR.



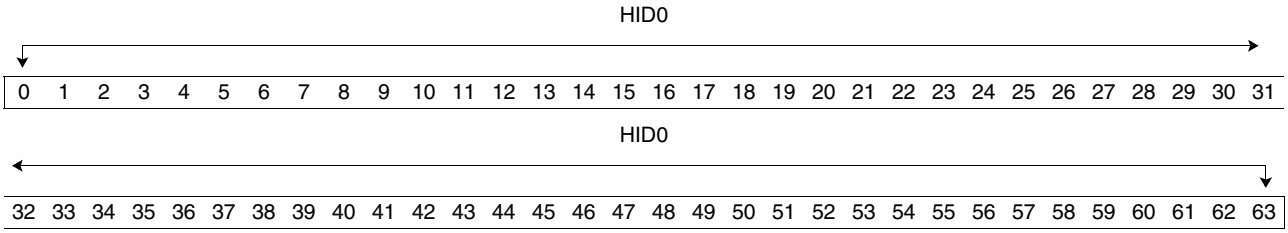
Bits	Field Name	Description
0:61	cmpr_addr	Word address to be compared.
62	brkpnt_en	Breakpoint enabled. An address match causes a trigger to the debug logic.
63	trans_en	Translation enabled. An IABR match is signaled only if MSR[IR] matches this bit.

Hardware Implementation Dependent Register 0 (HID0)

Address x'023101'

Type RW

Reset Reset to all zeros during POR.



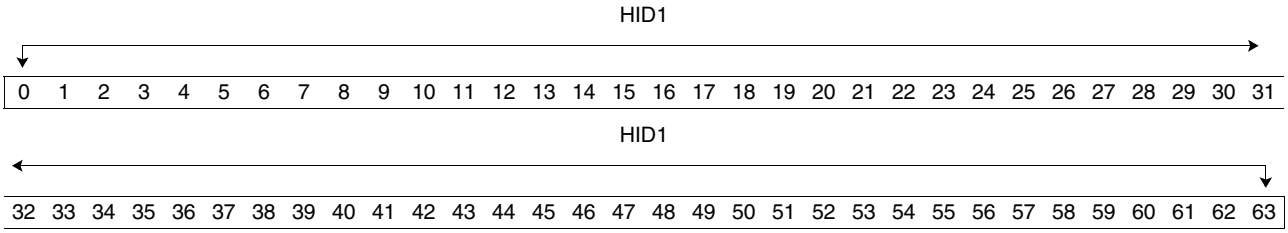
Bits	Field Name	Description
0:63	HID0	See the HID0 SPR definition in *Section 2.1.2.2 HID Registers (HID0, HID1, HID4, and HID5)* on page 56.

Hardware Implementation Dependent Register 1 (HID1)

Address x'023201'

Type RW

Reset Reset to all zeros during POR.



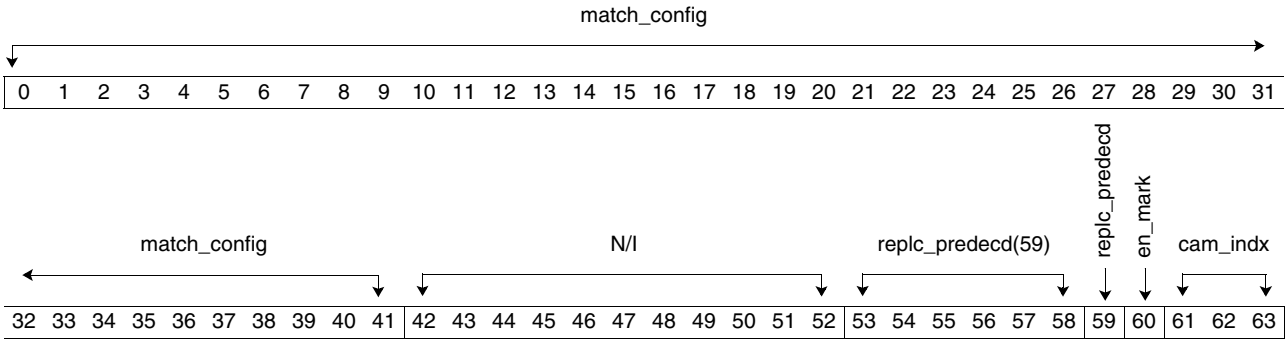
Bits	Field Name	Description
0:63	HID1	See the HID1 SPR definition in *Section 2.1.2.2 HID Registers (HID0, HID1, HID4, and HID5)* on page 56.

Instruction Match CAM (IMC) Register

Address x'023300'

Type WO

Reset N/A



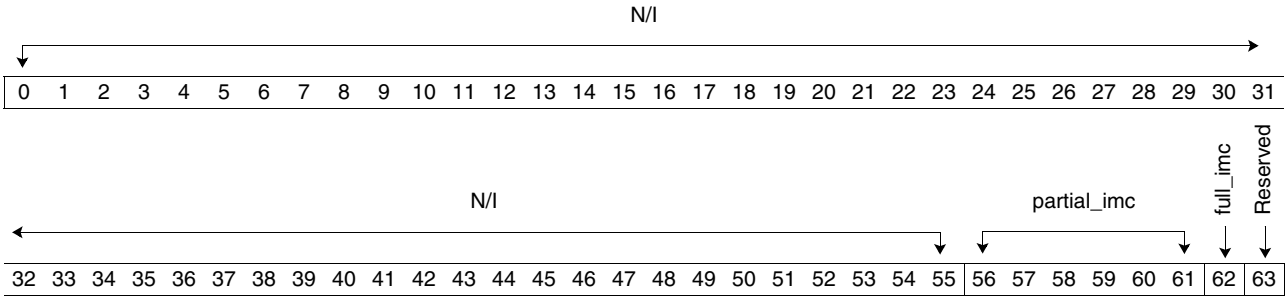
Bits	Field Name	Description
0:41	match_config	Match configuration bits. See bits 61:63 of the current register for definitions of the v0 and v1 bits. v0,v1 00 No match (disable entire entry). v0,v1 01 Match a '1'. v0,v1 10 Match a '0'. v0,v1 11 Match always (zeros and ones are irrelevant).
42:52	N/I	Not implemented.
53:58	replc_predecd(59)	Replacement predecode bits 0:5 (used only when bit 59 is set to '1'). **Note:** SPR cannot access this field.
59	replc_predecd	Replace predecode bits 0:5 with bits 53:58 of this register. **Note:** SPR cannot access this field.
60	en_mark	Enable mark (for power controller (SPU) sampling or enabling hardware workarounds). **Note:** Setting this bit is incompatible with Single-Step Trace Enable (MSR[SE]) mode. It will cause bits 35, 36, and 42 of Save/Restore Register 1 (SRR1), as well as the Sampled Instruction Address Register (SIAR) and the Sampled Data Address Register (SDAR), to be undefined.
61:63	cam_indx	Content addressable memory (CAM) index (entry select). 000:101 IMC[0:20] v0 IMC[21:41] v1 Partial match (for entries 0-5) Instruction bits 0:5, 21:31 Problem (PR), floating point (FP) available, Vector/SIMD Multimedia eXtension (VMX) Available mode 110 IMC[0:35] v0 IMC[36:41] '000000' Full match (v0 for entry 6) Instruction bits 0:31 PR, FP, VMX mode 111 IMC[0:35] v1 IMC[36:41] '000000' Full match (entry 7 = v1 for entry 6) Instruction bits 0:31 PR, FP, VMX mode

Patch Map (IMC Write Control Register)

Address x'023401'

Type RW

Reset Reset to all zeros during POR.



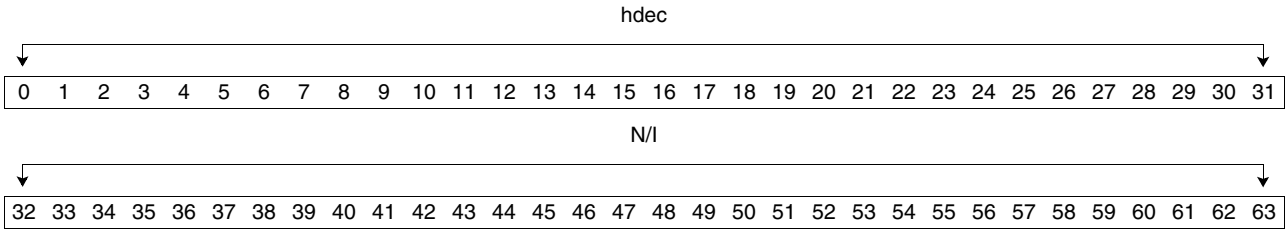
Bits	Field Name	Description
0:55	N/I	Not implemented
56:61	partial_imc	Partial match IMC entry (0:5) is being used for debug. Do not allow SPR write to corresponding IMC entry.
62	full_imc	Full match IMC entries (6 AND 7) are used for debug. Do not allow SPR write to corresponding IMC entries.
63	Reserved	Reserved.

Hypervisor Decrementer

Address x'023500'

Type RO

Reset Reset to '7FFFFFFF' during POR.



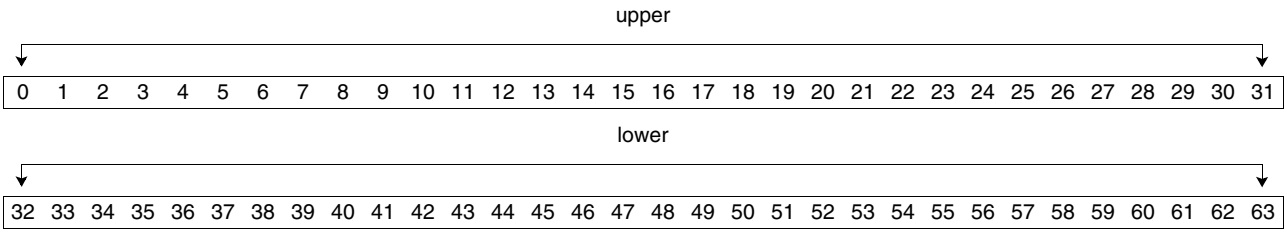
Bits	Field Name	Description
0:32	hdec	Current HDEC value. Continuously runs with 8:1 divided core clocks.
32:63	N/I	Not implemented.

Time Base Register

Address x'023600'

Type RO

Reset Reset to all zeros during POR.



Bits	Field Name	Description
0:32	upper	Upper time base. Runs with 8:1 divided core clocks or off an external nonuniform memory access (NUMA) oscillator.
32:63	lower	Lower time base. Runs with 8:1 divided core clocks or off an external NUMA oscillator.

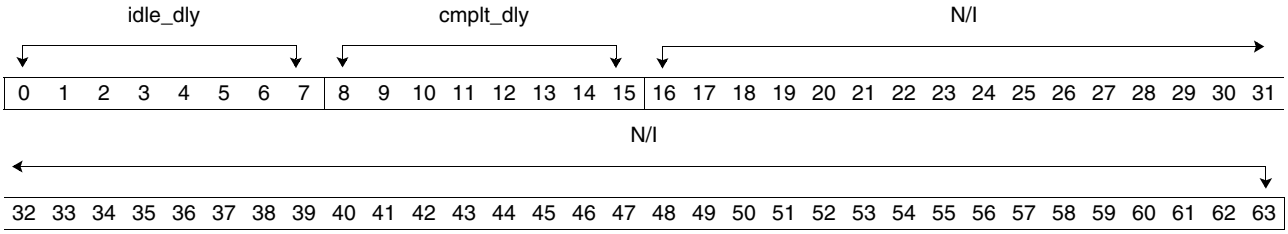
12.3.3 Processor Core Performance Monitor Sampling Control (x'02400X')

Performance Monitor Sampling Control Register

Address x'024001'

Type RW

Reset Reset to x'0414' during POR.



Bits	Field Name	Description
0:7	idle_dly	Idle delay.
8:15	cmplt_dly	Completion delay.
16:63	N/I	Not implemented.

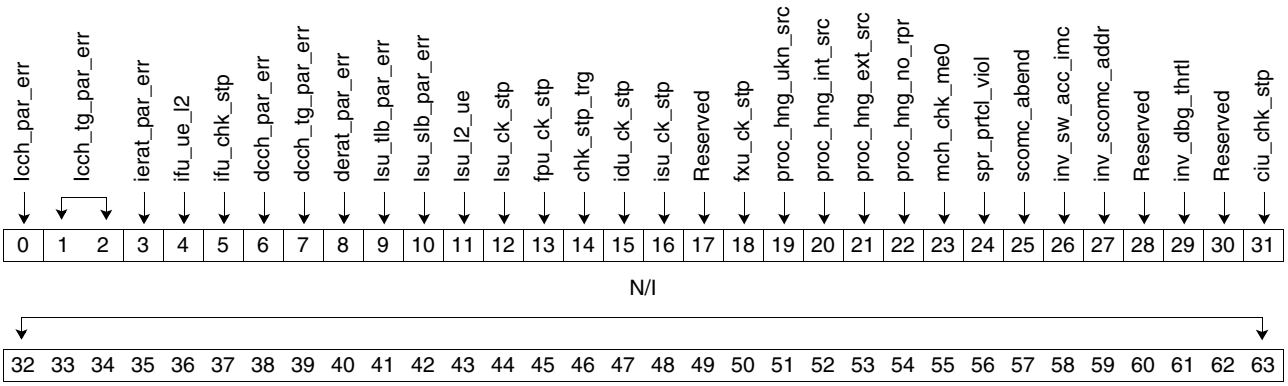
12.3.4 Processor Core FIR Facilities (x'03[0:5]XXX')

Core Fault Isolation Register
Core And-Mask Register
Core Or-Mask Register

Address x'030001'
x'031000' (AND)
x'032000' (OR)

Type RW
WO (And)
WO (OR)

Reset Reset to all zeros during POR.



Bits	Field Name	Description
0	lcch_par_err	I-cache parity error. Treated as an L1 cache miss. Instructions fetched from L2 or the prefetch buffer, and first instruction bypassed to decode. **Note:** Even with a hard failure in the I-cache, forward progress can be made because instructions are bypassed from the L2 to decode. Because the I-cache may report multiple errors for a soft fail, the operating system will be called to flush the I-cache. (Service Element Threshold)
1:2	lcch_tg_par_err	I-cache tag parity error (array 0, array 1). Treated as an L1 cache miss. Instructions fetched from L2 or the prefetch buffer, and first instruction bypassed to decode. **Note:** Even with a hard failure in the I-cache, forward progress can be made because instructions are bypassed from the L2 to decode. Because the I-cache may report multiple errors for a soft fail, the operating system will be called to flush the I-cache. (Service Processor Threshold)

Note: For processor hang detects, reference bits 26:31 of the CoreRAS Status Register to determine the extent of hang recovery that was required. In addition, for a processor hang with either an external or unknown source, the specific LSU and IFU status is available in bits 59:63 of the CoreRAS Status Register. Write to bits 55 and 41 respectively of the CoreRAS Control Register when clearing these processor hang FIR bits to clear the recovery and status information (if the hang was successfully recovered) for future diagnostics. Check bits 61:63 of the CoreRAS Mode Register to see what types of hang recovery are enabled (these settings affect how the Checkstop Enable mask should be set for these bits).

Bits	Field Name	Description
3	ierat_par_err	I-ERAT parity error. Treated as an I-ERAT miss. I-ERAT is reloaded from the TLB. Hardware does a flash invalidate of the I-ERAT. The hardware can tolerate one hard failure, but with slightly degraded performance because the error causes I-ERAT selection to go to the other processing unit where there will be a good I-ERAT. **Note:** A stuck fault on side 0 will hang the machine. (Service Processor Threshold)
4	ifu_ue_l2	The IFU fetched an uncorrected error (UE) from the L2. Corrupted data is discarded (not written into the I-cache). A speculative fetch that is not needed is discarded and instruction processing continues. An instruction that is required by the sequential execution model (SEM) causes a machine check interrupt. • SRR0 Address of corrupted instruction • SRR1 MSR bits and a bit indicating an instruction UE (PASSED ERROR)
5	ifu_chk_stp	The IFU detected a checkstop condition. Processor execution is halted. A system checkstop is raised that will halt the rest of the processors in the complex.
6	dcch_par_err	D-cache parity error. If the load was speculative and is eventually cancelled, instruction processing continues on the correct path (no interrupt is generated). If the load is required by the SEM, a machine check interrupt is generated reporting the error. • Common D-cache error recovery • SRR1 Bit indicating a LD/ST error • DSISR Bit indicating L1 data array parity (Machine Check Software Threshold)
7	dcch_tg_par_err	D-cache tag parity error. If the load was speculative and is eventually cancelled, instruction processing continues on the correct path (no interrupt is generated). If the load is required by the SEM, a machine check interrupt is generated reporting the error. • Common D-cache error recovery • SRR1 Bit indicating a LD/ST error • DSISR Bit indicating L1 address tag parity error
8	derat_par_err	D-ERAT parity error. If the access was speculative and is eventually cancelled, instruction processing continues on the correct path (no interrupt is generated). If the access is required by the SEM, a machine check interrupt is generated reporting the error. • Common D-cache error recovery • SRR1 Bit indicating a LD/ST error • DSISR Bit indicating D-ERAT parity (Machine Check Software Threshold)
9	lsu_tlb_par_err	LSU TLB parity error. If the access was speculative and is eventually cancelled, instruction processing continues on the correct path (no interrupt is generated), but the DERAT may be corrupted. If the access is required by the SEM, a machine check interrupt is generated reporting the error. • Common D-cache error recovery • SRR1 Bit indicating a LD/ST error • DSISR Bit indicating a TLB parity error (Machine Check Software Threshold)

Note: For processor hang detects, reference bits 26:31 of the CoreRAS Status Register to determine the extent of hang recovery that was required. In addition, for a processor hang with either an external or unknown source, the specific LSU and IFU status is available in bits 59:63 of the CoreRAS Status Register. Write to bits 55 and 41 respectively of the CoreRAS Control Register when clearing these processor hang FIR bits to clear the recovery and status information (if the hang was successfully recovered) for future diagnostics. Check bits 61:63 of the CoreRAS Mode Register to see what types of hang recovery are enabled (these settings affect how the Checkstop Enable mask should be set for these bits).

Bits	Field Name	Description
10	lsu_slb_par_err	LSU SLB parity error. If the access was speculative and is eventually cancelled, instruction processing continues on the correct path (no interrupt is generated), but the DERAT may be corrupted. If the access is required by the SEM, a machine check interrupt is generated reporting the error. • Common D-cache error recovery • SRR1 Bit indicating a LD/ST error • DSISR Bit indicating an SLB parity error If a multiple hit occurs, this bit is also set, because it cannot be determined whether there was a data parity error (because all the data is ORed together). (Machine Check Software Threshold)
11	lsu_l2_ue	LSU fetched an L2 UE. If the instruction is required by the SEM, a machine check interrupt is generated reporting the error. • SRR0 Address of the instruction to be executed • SRR1 Bit indicating a LD/ST error • DSISR Bit indicating the type of UE The I-ERAT may be corrupted. The TLB may be corrupted. (PASSED ERROR)
12	lsu_ck_stp	LSU detected checkstop condition. Processor execution is halted. A system checkstop is raised, which will halt the rest of the processors in the complex.
13	fpu_ck_stp	FPU 0 or FPU 1 detected a checkstop condition. Processor execution is halted. A system checkstop is raised, which will halt the rest of the processors in the complex.
14	chk_stp_trg	Checkstop on trigger (debug only). Programmable at the chip level. The trigger can be that processor execution is halted, a system checkstop, or even a clock stop for debug.
15	idu_ck_stp	IDU detected a checkstop condition. Processor execution is halted. A system checkstop is raised, which will halt the rest of the processors in the complex.
16	isu_ck_stp	ISU detected a checkstop condition. Processor execution is halted. A system checkstop is raised, which will halt the rest of the processors in the complex.
17	Reserved	Reserved (connected to additional ISU output).
18	fxu_ck_stp	FXU 0 or FXU 1 detected a checkstop condition. Processor execution is halted. A system checkstop is raised, which will halt the rest of the processors in the complex.
19	proc_hng_ukn_src	Processor hang detected with an unknown source (could be either internal or external). The source is not known for sure to be internal or external to the core because the LSU indication of outstanding transactions to the STS is not stable. A processor hang recovery will be invoked in an attempt to clear the hang condition only if recovery on unsure source has been enabled.
20	proc_hng_int_src	Processor hang detected that was due to an internal source. Source is known to be internal to the core because neither the IFU or LSU have outstanding demand requests to the STS. Processor hang recovery has been invoked in an attempt to clear the hang condition only if recovery on the internal source has been enabled.

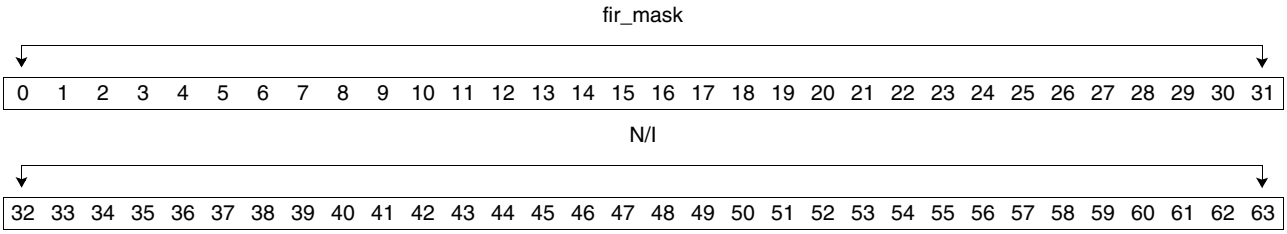
Note: For processor hang detects, reference bits 26:31 of the CoreRAS Status Register to determine the extent of hang recovery that was required. In addition, for a processor hang with either an external or unknown source, the specific LSU and IFU status is available in bits 59:63 of the CoreRAS Status Register. Write to bits 55 and 41 respectively of the CoreRAS Control Register when clearing these processor hang FIR bits to clear the recovery and status information (if the hang was successfully recovered) for future diagnostics. Check bits 61:63 of the CoreRAS Mode Register to see what types of hang recovery are enabled (these settings affect how the Checkstop Enable mask should be set for these bits).

Bits	Field Name	Description
21	proc_hng_ext_src	Processor hang detected that was due to an external source. Source is external to the core because the LSU or IFU indicates that a demand request to the STS is outstanding. Processor hang recovery has been invoked in an attempt to clear the hang condition only if recovery on the external source has been enabled. No field replaceable unit (FRU) call is possible; diagnostics should call for the next level of support.
22	proc_hng_no_rpr	Processor hung beyond repair. Core hang recovery failed. The processor is not making forward progress after all attempts at hang recovery. A system checkstop is raised, which will halt the rest of the processors in the complex. No FRU call is possible; diagnostics should call for the next level of support.
23	mch_chk_me0	Machine check and MSR[ME] equals '0'. Processor execution is halted. A system checkstop is raised, which will halt the rest of the processors in the complex. This is a latent checkstop. In other words, another error in the system caused a machine check. If this error is on by itself, diagnostics should call out the run time abstraction software (RTAS) or System Licensed Internal Code (SLIC).
24	spr_prtcl_viol	Core SPR bus has violated protocol or SCOMC arbiter error. Processor execution is halted. A system checkstop is raised, which will halt the rest of the processors in the complex. This is a software error. Diagnostics should call out the RTAS or SLIC.
25	scomc_abend	SCOMC ABEND. Either an SCOM reset was issued through JTAG or a core hang recovery was activated while an SCOMC request was active. This error is recoverable.
26	inv_sw_acc_imc	Invalid software access to IMC. The software, probably the performance monitor code, tried to write an IMC entry that was being used for debug (force only) according the patch map. The state of the machine is not altered.
27	inv_scomc_addr	Invalid SCOMC address. The address used by SCOMC was not accepted by any of the SCOM satellites. This error is recoverable. This is a software error. Diagnostics should call out the RTAS or SLIC.
28	Reserved	Spare. System checkstop.
29	inv_dbg_thrtl	Service processor attempted an invalid debug throttle setting. Command is ignored.
30	Reserved	Spare. System checkstop.
31	ciu_chk_stp	CIU detected checkstop condition. Processor execution is halted. A system checkstop is raised, which will halt the rest of the processors in the complex.
32:63	N/I	Not implemented.

Note: For processor hang detects, reference bits 26:31 of the CoreRAS Status Register to determine the extent of hang recovery that was required. In addition, for a processor hang with either an external or unknown source, the specific LSU and IFU status is available in bits 59:63 of the CoreRAS Status Register. Write to bits 55 and 41 respectively of the CoreRAS Control Register when clearing these processor hang FIR bits to clear the recovery and status information (if the hang was successfully recovered) for future diagnostics. Check bits 61:63 of the CoreRAS Mode Register to see what types of hang recovery are enabled (these settings affect how the Checkstop Enable mask should be set for these bits).

Core Fault Isolation Mask Register
Core And-Mask Register
Core Or-Mask Register

Address	x'030400' x'031401' (AND) x'032401' (OR)
Type	RW WO (AND) WO (OR)
Reset	Reset to all ones during POR.



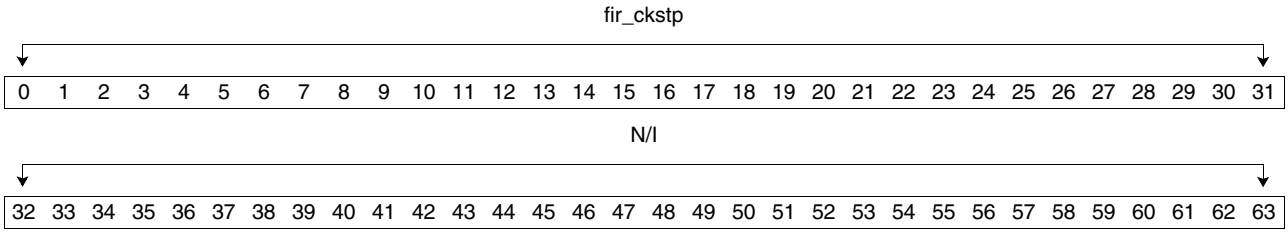
Bits	Field Name	Description
0:31	fir_mask	Mask 1 FIR bit masked off.
32:63	N/I	Not implemented.

Core Checkstop Enable Registers

Address x'030800'

Type RW

Reset Reset to all zeros during POR.



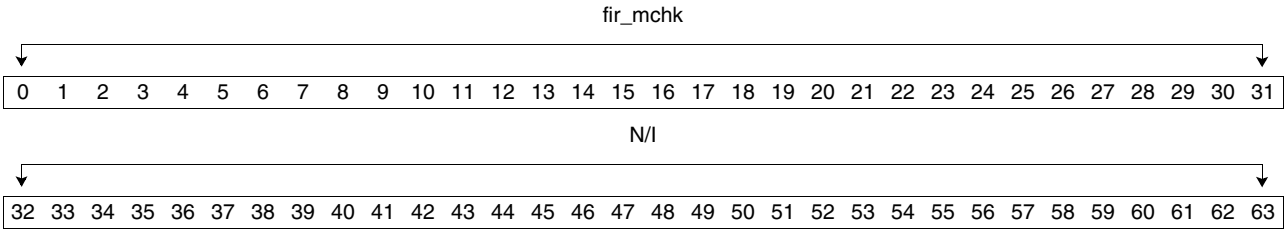
Bits	Field Name	Description
0:31	fir_ckstp	Causes a checkstop to occur when the corresponding FIR bit is set.
32:63	N/I	Not implemented.

Core Machine-Check Enable Register

Address x'030901'

Type RW

Reset Reset to all zeros during POR.



Bits	Field Name	Description
0:31	fir_mchk	Causes a machine check to occur when the corresponding FIR bit is set.
32:63	N/I	Not implemented.

12.3.5 Instruction Mark Configuration (x'03600X')

Instruction Mark Configuration Register

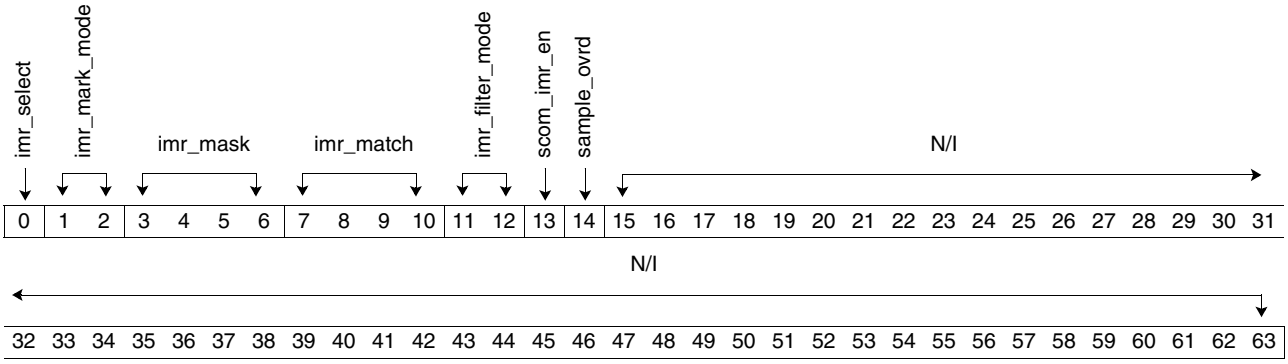
Address x'036001'

Type RW

Reset Reset to nonmarking mode during POR.

imr_mask '1111'
imr_select '1'

All other bits are zero at POR.



Bits	Field Name	Description
0	imr_select	Determine stage-1 eligibility from: 0 Predecode match (uses imr_match and imr_mask, bits 3:10 below). 1 Instruction match (IMR bit from IMC match).
1:2	imr_mark_mode	Chooses which instructions are stage-2 eligible for marking based on: 00 All stage-1, eligible internal operations (IOPs). 01 Only stage-1, eligible IOPs that resulted from microcode expansion. 10 Only one IOP per PowerPC instruction regardless of stage-1 eligibility. 11 First IOP that goes to the LSU for every PowerPC LD/ST instruction regardless of stage-1 eligibility.
3:6	imr_mask	Value ANDed with predecode before performing the predecode match.
7:10	imr_match	Value used to perform an exact compare against the masked 4-bit predecode field, used in determining stage 1 eligibility. To predecode, match *all* IOPs; set imr_mask to '0000' and imr_match to '0000'.
11:12	imr_filter_mode	Picks which stage 2, eligible IOPs are sampled as a marked group in the machine. 00 Sample all stage-2 eligible IOPs. 01 Sample only the first stage-2 eligible IOP in each group. 10 Randomly sample from all stage-2 eligible IOPs. 11 Sample only the first randomly picked stage-2 eligible IOP in each group. **Note:** Used for both Performance Monitor and Debug modes.
13	scom_imr_en	SCOM IMR enable. Use the first eleven bits of this register instead of performance monitor selects from the Monitor Mode Control Register 2 (MMCR2).
Usage Note: To use the IMC facility to set marks for debug triggering based on instruction match, set this register to "80060000 00000000" after configuring the Instruction Match CAM (IMC) Register (x'023300').		

Bits	Field Name	Description
14	sample_ovrd	Sample override. Overrides the active performance monitor "*ok_to_sample*" indication. Enables the above marking modes for use in debug.
15:63	N/I	Not implemented.

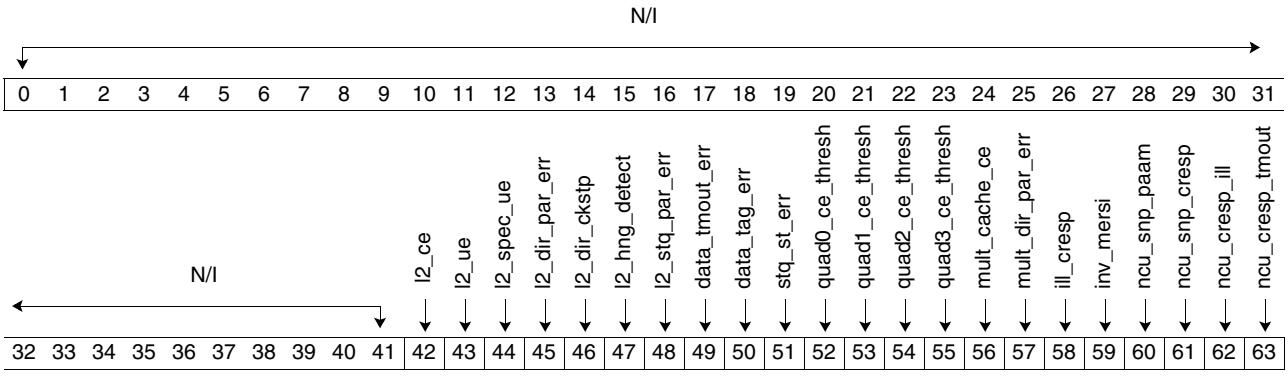
Usage Note: To use the IMC facility to set marks for debug triggering based on instruction match, set this register to "80060000 00000000" after configuring the Instruction Match CAM (IMC) Register (x'023300').

12.4 Storage Subsystem SCOM Register Definition

12.4.1 L2 SCOM Register Definition

L2 Fault Isolation Register
L2 Fault Isolation And-Mask Register
L2 Fault Isolation Or-Mask Register
L2 Fault Isolation Checkstop Register
L2 Error Mask Register
L2 Error And-Mask Register
L2 Error Or-Mask Register
L2 Checkstop Enable

FIR Address	x'040000' x'041001' (AND) x'042001' (OR)
Error Mask Address	x'040401' x'041400' (AND) x'042400' (OR)
Checkstop Enable Address	x'040801'
Type	RW (FIR) WO (And) WO (OR) WR (Checkstop)
Reset	Reset to all zeros during POR.



Bits	Field Name	Description	Default Mask Settings	
			Error Mask	Checkstop Enable
0:41	N/I	Not implemented	0	0

Bits	Field Name	Description	Default Mask Settings	
			Error Mask	Checkstop Enable
42	l2_ce	L2-cache correctable data error (CE). • L2 CE on processor request (instruction or data). Data is corrected before forwarding to the processor. Corrected data is written back to the L2 cache. • L2 CE on castout (flush, read with intent to modify [RWITM], or victim of a replacement). Data is corrected before forwarding to the BIU. The threshold equals 32.	0	0
43	l2_ue	L2-cache uncorrectable error (UE). • A UE is detected in the L2-cache response to a processor load request. The UE response is sent to the requesting core. • A store hits in the L2 line that contains a UE. The store is merged into the line (timing does not permit discarding the store). Write a UE error checking and correction (ECC) to distinguish between a passed error from another source and a local UE that has been altered. • Uncorrectable error detected in the L2 cache in response to a cast out operation. A special UE (SUE) is sent to memory.	0	1
44	l2_spec_ue	L2-cache special UE. Same as bit 43, cache UE (passed error).	1	0
45	l2_dir_par_err	L2-directory parity error. The failing directory is refreshed with the contents of the other directory. The threshold equals 32.	0	0
46	l2_dir_ckstp	L2-directory checkstop. L2-directory parity error is detected on both halves of the directory. System checkstop.	0	1
47	l2_hng_detect	L2 hang detect. No response to memory read request. The same action is taken regardless of whether some other component acknowledged the request and then never returned the data, or no component on the fabric acknowledged the request. System checkstop.	0	1
48	l2_stq_par_err	L2 store-queue parity error. Any of the four store queues has a parity error. System checkstop.	0	1
49	data_tmout_err	Read/claim (RC) machine data timeout error. This is an internal control error. System checkstop.	0	1
50	data_tag_err	Non-cacheable control or RC machine data tag error. Either the non-cacheable control (ncctl) or the RC final state machine (rcfsm0-3) detected a valid data tag match when they were not expecting data. This is an internal control error. System checkstop.	0	1
51	stq_st_err	Store-queue store error. • A store command was interleaved between the first and second store-queue write/store (stqw_seq_err). or • A store-queue request was detected when the 4-entry store queue was full (stq_overflow_err).	0	1
52	quad0_ce_thresh	Quadrant0 CE threshold. Multiple cache CEs have been detected during a hang pulse duration.	0	0
53	quad1_ce_thresh	Quadrant1 CE threshold. Multiple cache CEs have been detected during a hang pulse duration.	0	0

Bits	Field Name	Description	Default Mask Settings	
			Error Mask	Checkstop Enable
54	quad2_ce_thresh	Quadrant2 CE threshold. Multiple cache CEs have been detected during a hang pulse duration.	0	0
55	quad3_ce_thresh	Quadrant3 CE threshold. Multiple cache CEs have been detected during a hang pulse duration.	0	0
56	mult_cache_ce	Multiple cache CEs. Multiple cache CEs have been detected during a hang pulse duration from more that one quadrant.	0	0
57	mult_dir_par_err	Multiple directory parity errors. Multiple directory parity errors were detected within a period of two hang pulses. This indicates that the array error is not an intermittent fault. Because the BIU cannot make forward progress with a stuck fault in the directory, the system will be checkstopped.	0	1
58	ill_cresp	Illegal CRESP seen in receive state machine (RCFSM). The RC machine detected an illegal snoop, a combined response (CRESP) operation.	0	1
59	inv_mersi	Invalid MERSI detected. The RC machine detected or wrote an invalid modified/exclusive/reserved/shared/invalid (MERSI) cache-coherency protocol state in the directories.	0	1
60	ncu_snp_paam	NCU snoop PAAM error checkstop. This error occurs when the NCU snooper detects another ICBI or translation lookaside buffer invalidate entry (TLBI) request while the previous request is still active. That is, CRESP has not returned yet. The illegal previous adjacent address match (PAAM) request will be lost because the snooper does not register it; hence, there is *no* snoop response.	0	1
61	ncu_snp_cresp	NCU snoop CRESP error checkstop. This error occurs when an NCU snooper detects illegal values on CRESP(0:3): 0000 Good xxx1 Retry other Illegal The errors are registered, and the snooper hangs while waiting for a good or retry response. A secondary PAAM error can occur as a result. A debug switch, $\overline{\text{cfg\_snp\_cresp\_ckstp\_en}}$, is provided to turn error detection off. In this case, the snooper will hang, and the result is undermined.	0	1
62	ncu_cresp_ill	NCU CRESP illegal error. This error occurs when the NCU receives a cresp_valid from the BIU with a CRESP(0:3) value that is illegal based on the transaction sent.	0	1
63	ncu_cresp_tmout	NCU CRESP timeout error. This error occurs when any of the state machines sourcing a transaction (store, load, or a micro-operation) to the bus time out while waiting for a valid and good CRESP. The timeout occurs upon the second hang pulse seen while waiting for the good CRESP.	0	1

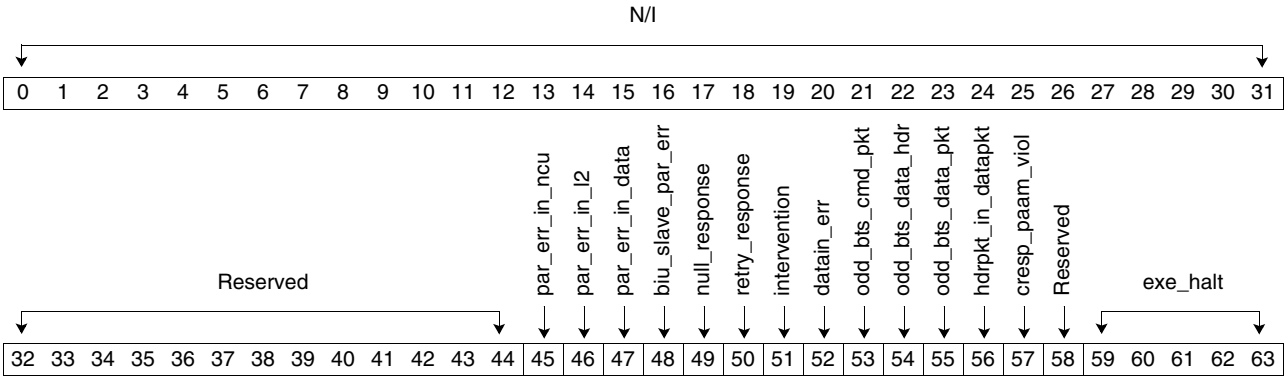
12.4.2 BIU SCOM Register Definition

BIU Fault Isolation Register/And-Mask/Or-Mask

Address x'0A0001'

Type RW (FIR)
WO (AND)
WO (OR)

Reset Reset to all zeros during POR.



Bits	Field Name	Description	Error Mask	Checkstop Enable
0:31	N/I	Not implemented.	—	—
32:44	Reserved	Spare.	—	—
45	par_err_in_ncu	Parity error for input from NCU. Processor execution is halted. A system checkstop is raised, which will halt the rest of the processors in the complex.	0	1
46	par_err_in_l2	Parity error for input from L2. Processor execution is halted. A system checkstop is raised, which will halt the rest of the processors in the complex.	0	1
47	par_err_in_data	Parity error for input from encoded data. Processor execution is halted. A system checkstop is raised, which will halt the rest of the processors in the complex.	0	1
48	biu_slave_par_err	A BIU slave reports a parity error using a transaction response. Processor execution is halted. A system checkstop is raised, which will halt the rest of the processors in the complex.	0	1
49	null_response	A transaction response returns null. Processor execution is halted. A system checkstop is raised, which will halt the rest of the processors in the complex.	0	1
50	retry_response	A bus slave returns a transaction response retry on a data packet. Processor execution is halted. A system checkstop is raised, which will halt the rest of the processors in the complex.	0	1
51	intervention	A bus slave returns an intervention without shared or modified, or returns shared, modified, or intervention on a castout. Processor execution is halted. A system checkstop is raised, which will halt the rest of the processors in the complex.	0	1
52	datain_err	An error occurs on bus input data when encode is disabled. Processor execution is halted. A system checkstop is raised, which will halt the rest of the processors in the complex.	0	1

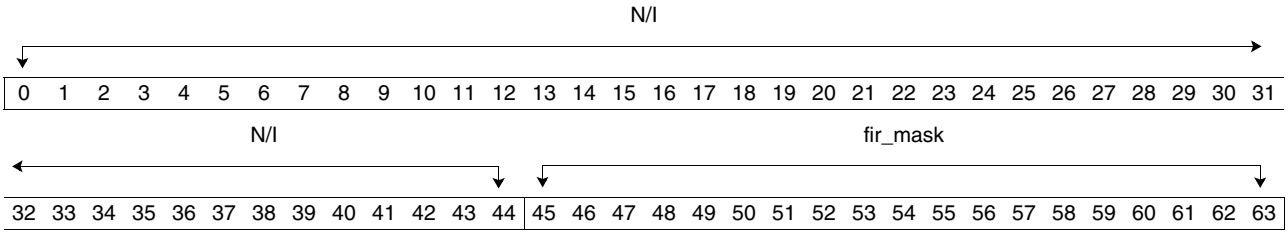
Bits	Field Name	Description	Error Mask	Checkstop Enable
53	odd_bts_cmd_pkt	The command packet is an odd number of beats. Processor execution is halted. A system checkstop is raised, which will halt the rest of the processors in the complex.	0	1
54	odd_bts_data_hdr	The data header packet is an odd number of beats. Processor execution is halted. A system checkstop is raised, which will halt the rest of the processors in the complex.	0	1
55	odd_bts_data_pkt	The data packet is an odd number of beats. Processor execution is halted. A system checkstop is raised, which will halt the rest of the processors in the complex.	0	1
56	hdrpkt_in_datapkt	A header packet appears inside a data packet. Processor execution is halted. A system checkstop is raised, which will halt the rest of the processors in the complex.	0	1
57	cresp_paam_viol	CRESP PAAM window violation. Processor execution is halted. A system checkstop is raised, which will halt the rest of the processors in the complex.	0	1
58	Reserved	Spare.	—	—
59:63	exe_halt	Tag associated with transaction response in reported errors. Processor execution is halted. A system checkstop is raised, which will halt the rest of the processors in the complex.	0	1

BIU Error Mask/And-Mask/Or-Mask

Address x'0A0400'

Type RW
WO (AND)
WO (OR)

Reset Reset to ones during POR.



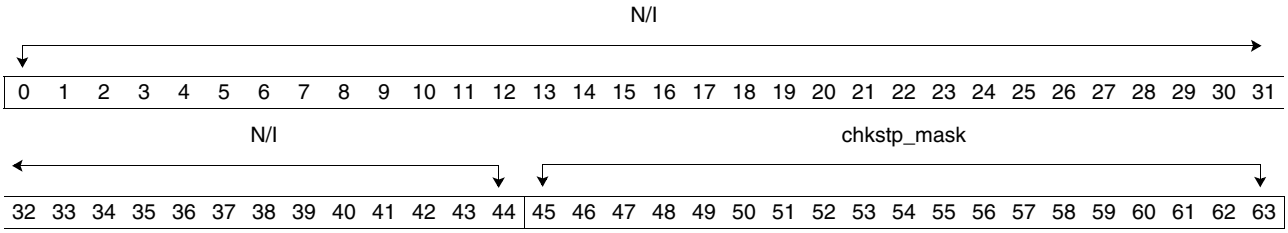
Bits	Field Name	Description
0:44	N/I	Not implemented.
45:63	fir_mask	Mask. 1 FIR bit is masked off.

BIU Checkstop Enable

Address x'0A0800'

Type RW

Reset Reset to zeros during POR.



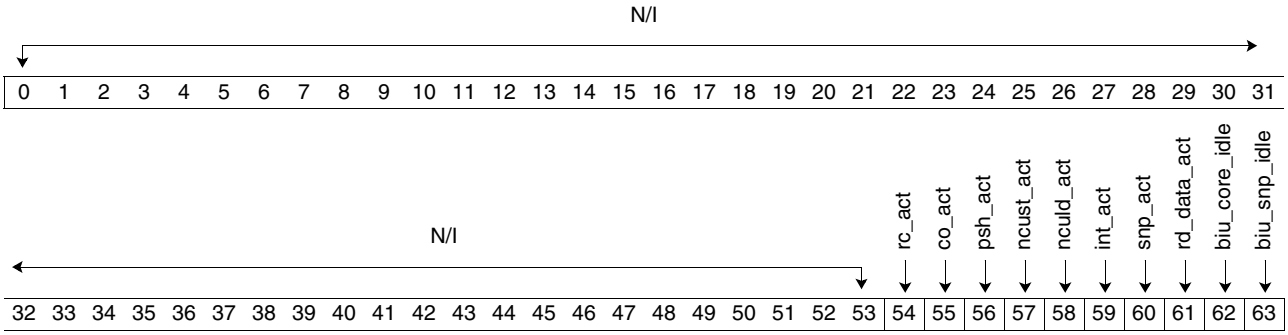
Bits	Field Name	Description
0:44	N/I	Not implemented.
45:63	chkstp_mask	Mask. 1 Checkstop.

BIU Status Register

Address x'0A9000'

Type RW

Reset Reset to all zeros during POR.



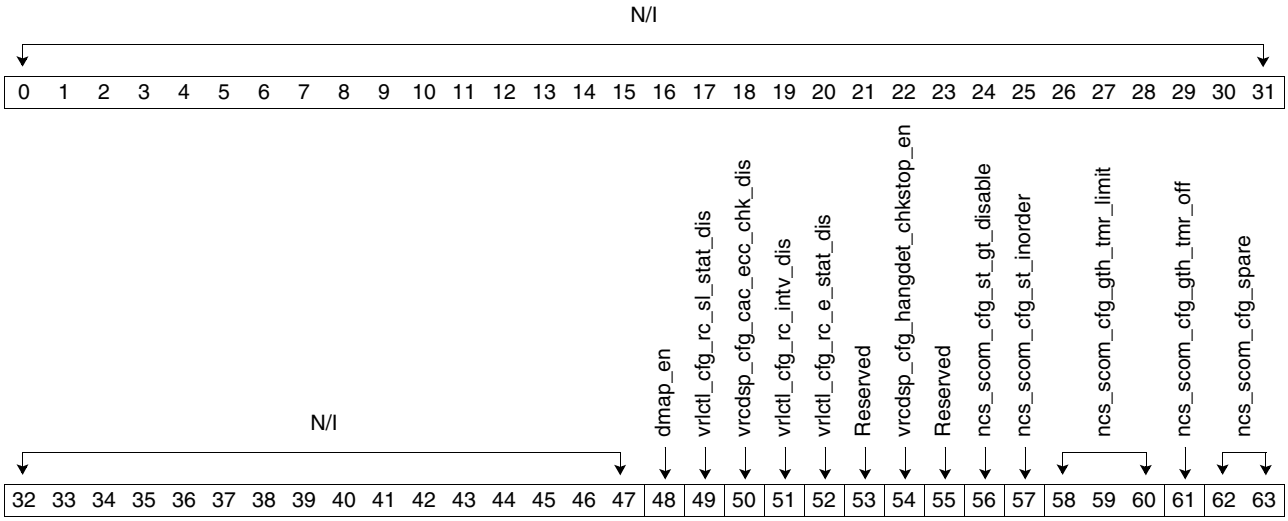
Bits	Field Name	Description
0:53	N/I	Not implemented.
54	rc_act	RC is the active operation (bfb_status_decode[0:3] equals '0000' or '1000').
55	co_act	Cast Out (CO) is the active operation (bfb_status_decode[0:3] equals '0001' or '1001').
56	psh_act	Cache push on snoop response (PSH) is the active operation (bfb_status_decode[0:3] equals '0010' or '1010').
57	ncust_act	Noncacheable unit store (NCUST) is the active operation (bfb_status_decode[0:3] equals '0011' or '1011').
58	nculd_act	Noncacheable unit load (NCULD) is the active operation (bfb_status_decode[0:3] equals '0100' or '1100').
59	int_act	INT is the active operation (bfb_status_decode[0:3] equals '0101' or '1101').
60	snp_act	The snoop command is the active operation (bfb_status_decode[0:3] equals '0110' or '1110').
61	rd_data_act	Read data is the active operation (bfb_status_decode[0:3] equals ['0111' to '1110']).
62	biu_core_idle	The BIU core is idle.
63	biu_snp_idle	The BIU snoop is idle.

BIU Mode Register

Address x'043000'

Type RW

Reset Reset to all zeros during POR.



Bits	Field Name	Description
0:31	N/I	Not implemented.
32:47	N/I	Not implemented.
48	dmap_en	When set to '1', puts the L2 cache into Direct Mapped mode so that victim selection is based on a straight decode of address bits 42:44.
49	vrlctl_cfg_rc_sl_stat_dis	v_rcfsm disables going to the Shared Invalid (SI) state. Instead, it goes to Shared (S).
50	vrcdsp_cfg_cac_ecc_chk_dis	Disables viewing of any error correction code check (ECCCK) error detection.
51	vrlctl_cfg_rc_intv_dis	v_rctag: 1 Forces the intervention bit off.
52	vrlctl_cfg_rc_e_stat_dis	v_rcfsm: 1 Forces all Exclusive (E) directory states to Shared (S). 0 (Default) Allows E state.
53	Reserved	Spare.
54	vrcdsp_cfg_hangdet_chkstop_en	Enables a livelock warning signal to be sent to the L2 slice macro, where it can cause a checkstop.
55	Reserved	Spare.
56	ncs_scom_cfg_st_gt_disable	NCU store gather control disable.
57	ncs_scom_cfg_st_inorder	Force g to '0'. Stores go out to the bus in order. Hence, one instruction has to complete with no retry before the other instruction can go out.

Bits	Field Name	Description
58:60	ncs_scom_cfg_gth_tmr_limit	These bits set the value for the timeout counter for the gather logic. The timeout counter is programmable for gather fine tuning. Setting these bits to zeros, in effect, discourages gathering although it does not totally disable it. Its value is (^bit0 & ^bit1 & bit2 & 0 & 0). The timeout is set as default '11000' pclk count when the latches are set to '000'. Its range is (0:56 pclks, step by 4).
61	ncs_scom_cfg_gth_tmr_off	Turn off gather timer. No timeout for gather.
62:63	ncs_scom_cfg_spare	Spare.

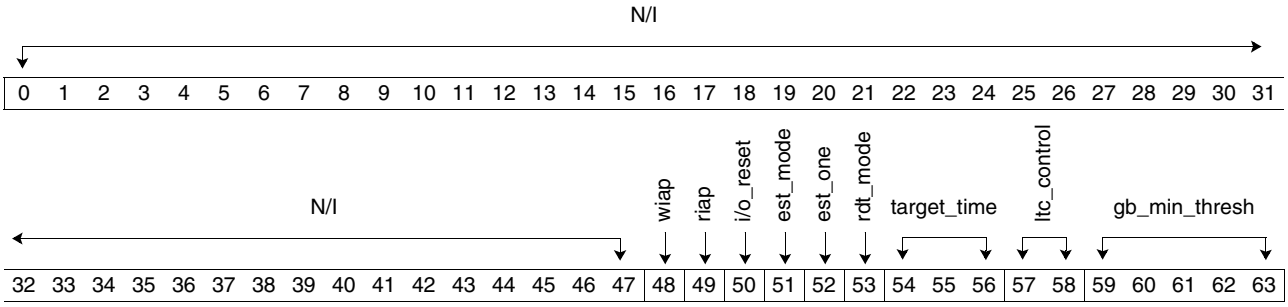
12.4.3 Processor Interconnect Registers

PI Mode Register 0

Address x'083000'

Type R/W

Reset Reset to all zeros during POR.



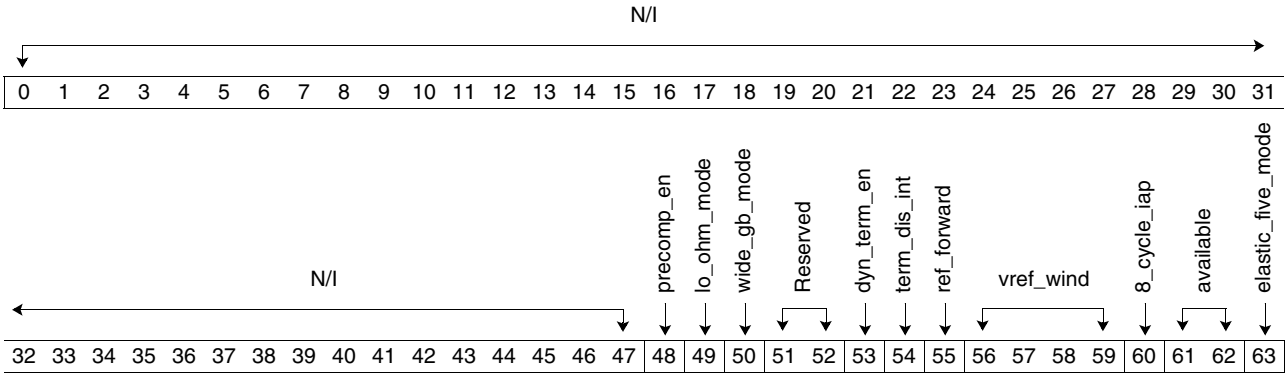
Bits	Field Name	Description
0:47	N/I	Not implemented.
48	wiap	Starts the IAP or self test on the driver.
49	riap	Starts the IAP or self test on the receiver.
50	i/o_reset	Resets the receiver, and readies for the IAP.
51	est_mode	Selects the shorts/open test.
52	est_one	Selects the polarity of the shorts test. 0 Creates a walking ones pattern 1 Creates a walking zeros pattern
53	rdt_mode	Selects random data test (RDT) mode.
54:56	target_time	Sets the target time.
57:58	ltc_control	Learned target cycle (LTC). 00 Minimum 01 Minimum + 1 10 Minimum + 2 11 Don't use LTC
59:63	gb_min_thresh	Guardband minimum threshold.

PI Mode Register 1

Address x'083101'

Type R/W

Reset Reset to all zeros during POR.



Bits	Field Name	Description
0:47	N/I	Not implemented.
48	precomp_en	Enables Precompensation mode at the drivers. Note: Bit 48 and bit 49 work together to determine how precompensation works. Bit 48 Bit 49 0 0 Precompensation is not enabled; high impedance is selected. 1 0 Precompensation is enabled; high impedance is selected. Precompensation may improve intersymbol interference. 0 1 Precompensation is not enabled; low impedance is selected. 1 1 Precompensation is enabled; low impedance is selected. Although precompensation is enabled, it has no effect when the drivers are in low-impedance mode.
49	lo_ohm_mode	Enables low Ω mode at the drivers.
50	wide_gb_mode	Defines the meaning of the double guardband fail calculation. 0 Enables a narrow mode where a double guardband fail can occur if the bit fails a setup and a hold test at any time. 1 Enables a wide mode where a double guardband fail only occurs if the bit fails a setup and a hold for the same data beat.
51:52	Reserved	Not used but available.
53	dyn_term_en	Switches between clamp and termination on.
54	term_dis_int	Disables termination on the receivers.
55	ref_forward	Apply a reference voltage (V_{REF}) to the receivers. Calculated from the swing in the clock receiver.
56:59	vref_wind	Signed number used to offset V_{REF}.
60	8_cycle_iap	Selects IAP pattern length (4 or 8). 0 4 1 8
61:62	available	Not used but available.
63	elastic_five_mode	This special mode of the PI is not supported.

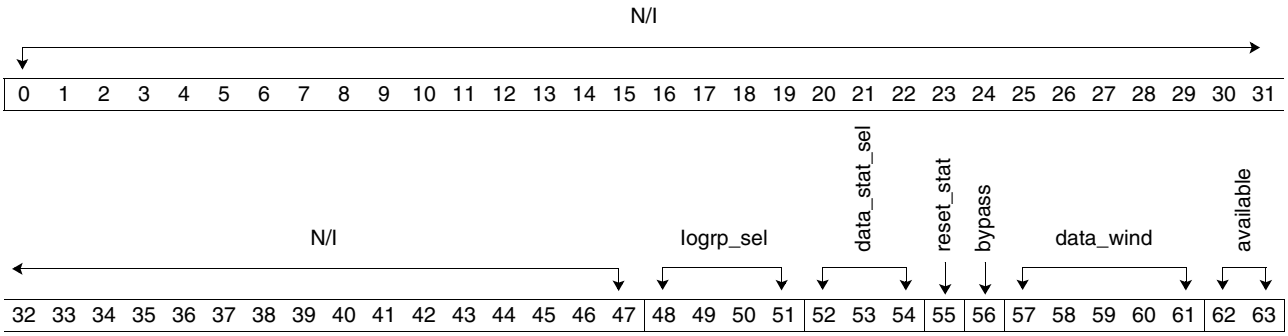
PI Mode Register 2

Address x'083201'

Type R/W

Note: Do not write during functional mode.

Reset Reset to all zeros during POR.



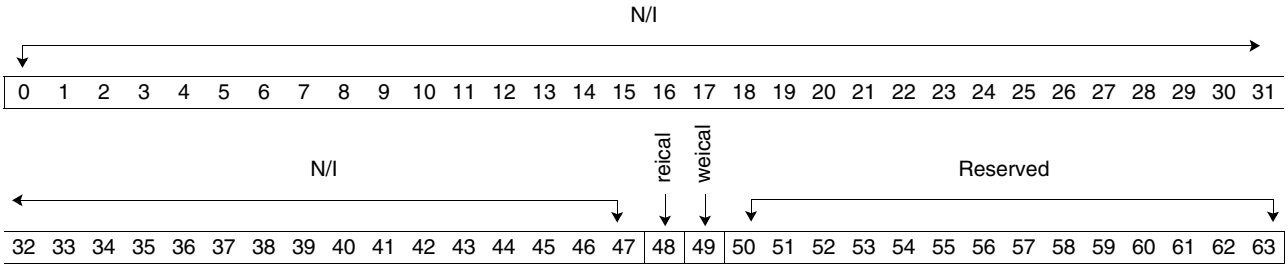
Bits	Field Name	Description
0:47	N/I	Not implemented.
48:51	logrp_sel	Selects the clock group for the status register.
52:54	data_stat_sel	Selects information to be retrieved for the status register.
55	reset_stat	Resets status information (stat[0:8] in the PI Status Register). Also begins a learn target cycle command.
56	bypass	Puts the receiver in bypass mode (no bit alignment and target cycle). For the second generation of PI (PI2), the PI bypass is not implemented the same way as for the first generation of PI (PI1).
57:61	data_wind	Add or subtract a delay from each bit. (Dissimilar to PI1).
62:63	available	Not used but available.

PI Mode Register 3

Address x'083300'

Type WO

Reset Reset to all zeros during POR.



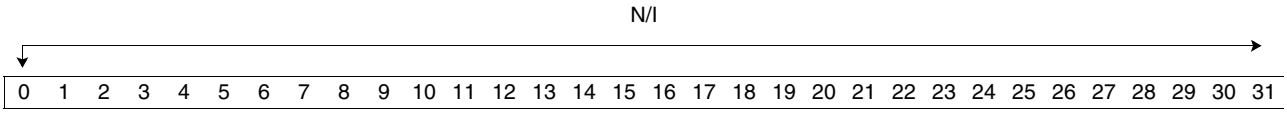
Bits	Field Name	Description
0:47	N/I	Not implemented.
48	reical	Enables processor interface recalibration (EICAL) for the receivers.
49	weical	Enables EICAL for the drivers.
48:63	Reserved	Reserved; not implemented.

PI Status Register

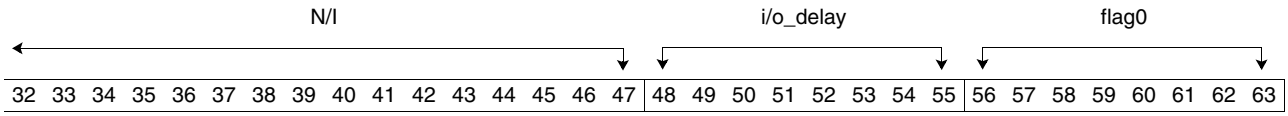
Address x'084001'

Type RO

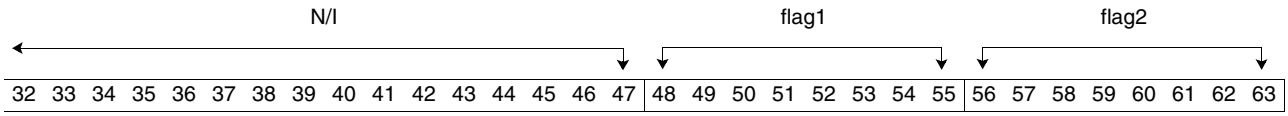
Reset Reset to all zeros during POR.



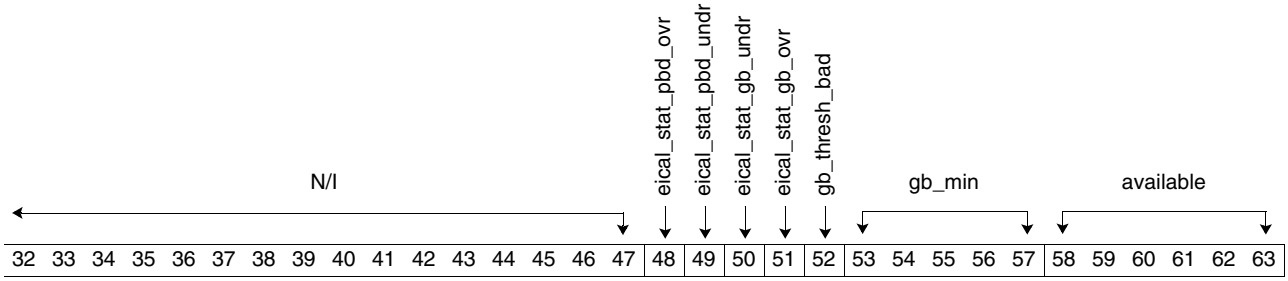
iogrp_sel = '0000' and data_stat_sel = '000'



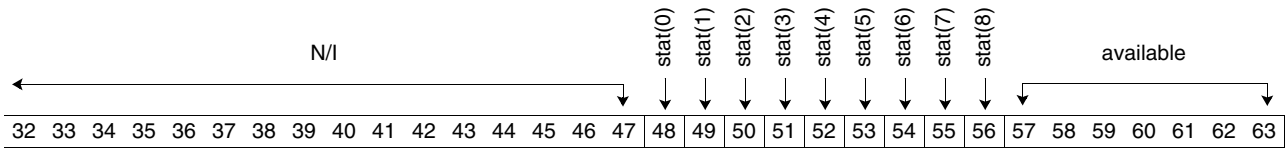
iogrp_sel = '0000' and data_stat_sel = '001'



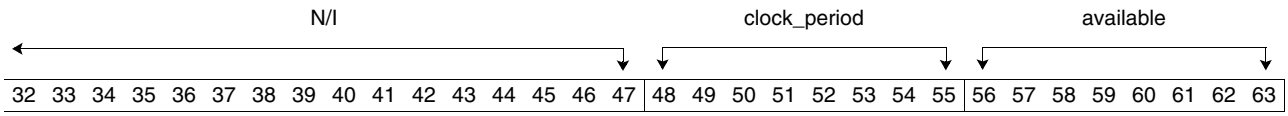
iogrp_sel = '0000' and data_stat_sel = '010'



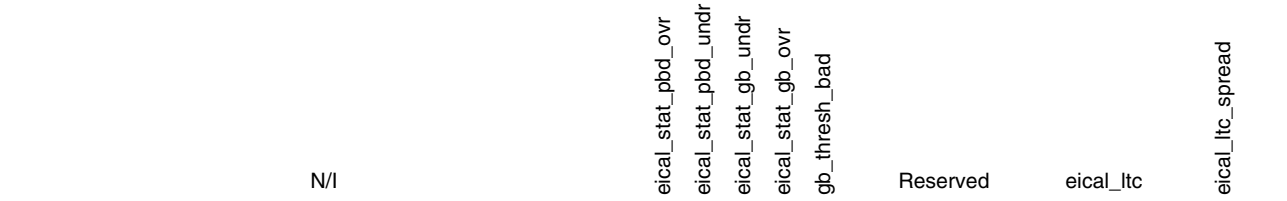
iogrp_sel = '0000' and data_stat_sel = '011'

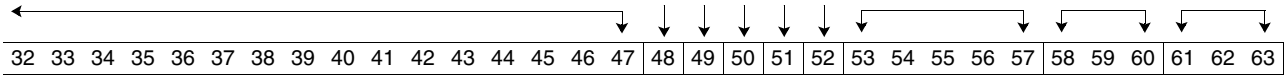


iogrp_sel = '1111' and data_stat_sel = '000'

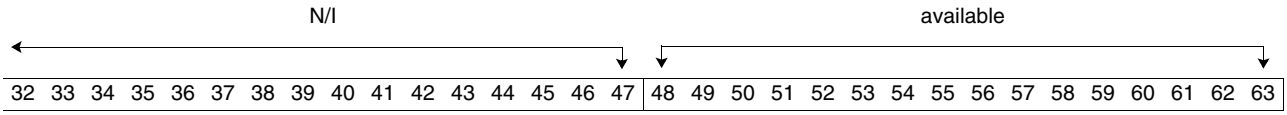


iogrp_sel = '1111' and data_stat_sel = '001'

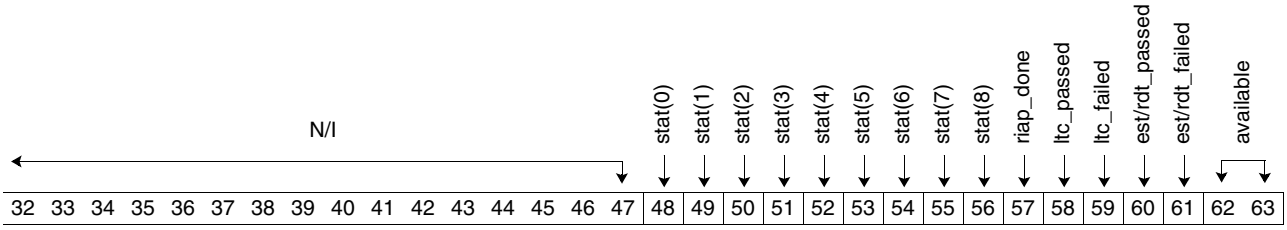




iogrp_sel = '1111' and data_stat_sel = '011'



iogrp_sel = '1111' and data_stat_sel = '111'



Bits	Field Name	Description
If iogrp_sel = '0000' and data_stat_sel = '000' (Set in Mode Register 2)		
0:47	N/I	Not implemented.
48:55	i/o_delay	Number of delay elements in clock path.
56:63	flag0	Number of delay elements for flag 0.
If iogrp_sel = '0000' and data_stat_sel = '001'		
0:47	N/I	Not implemented.
48:55	flag1	Number of delay elements for flag 1.
56:63	flag2	Number of delay elements for flag 2.
If iogrp_sel = '0000' and data_stat_sel = '010'		
0:47	N/I	Not implemented.
48	eical_stat_pbd_ovr	Overflow of per-bit-deskew counter during IAP or EICAL.
49	eical_stat_pbd_ undr	Underflow of per-bit-deskew counter during IAP or EICAL.
50	eical_stat_gb_undr	Guardband underflow during IAP or EICAL.
51	eical_stat_gb_ovr	Guardband overflow during IAP or EICAL.
52	gb_thresh_bad	Minimum guardband less than minimum required guardband.
53:57	gb_min	Minimum guardband value.
58:63	available	Not used but available.
If iogrp_sel = '0000' and data_stat_sel = '011'		
0:47	N/I	Not implemented.
48	stat(0)	Clock period too large to be measured with delay elements available.
49	stat(1)	IAP pattern '1' could not be found.
50	stat(2)	Right edge of data eye not found.
51	stat(3)	PBD maximum reached in one or more channels.
52	stat(4)	Failed to find flag 0.

Bits	Field Name	Description
53	stat(5)	Failed to find flag 1.
54	stat(6)	Failed to find flag 2.
55	stat(7)	Final clock delay does not find IAP pattern.
56	stat(8)	Negative I/O clock delay.
57:63	available	Not used but available.
If iogrp_sel = '1111' and data_stat_sel = '000'		
0:47	N/I	Not implemented.
48:55	clock_period	Clock period expressed in delay elements.
56:63	available	Not used but available.
If iogrp_sel = '1111' and data_stat_sel = '001'		
0:47	N/I	Not implemented.
48	eical_stat_pbd_ovr	Overflow of per-bit-deskew counter during IAP or EICAL.
49	eical_stat_pbd_ undr	Underflow of per-bit-deskew counter during IAP or EICAL.
50	eical_stat_gb_undr	Guardband underflow during IAP or EICAL.
51	eical_stat_gb_ovr	Guardband overflow during IAP or EICAL.
52	gb_thresh_bad	Minimum guardband less than minimum required guardband.
53:57	Reserved	Not used but available.
58:60	eical_ltc	Learned target cycle value.
61:63	eical_ltc_spread	Difference in LTC between clock groups.
If iogrp_sel = '1111' and data_stat_sel = '011'		
0:47	N/I	Not implemented.
48:63	available	Not used but available.
If iogrp_sel = '1111' and data_stat_sel = '111'		
0:47	N/I	Not implemented.
48	stat(0)	Clock period too large to be measured with delay elements available.
49	stat(1)	IAP pattern '1' could not be found.
50	stat(2)	Right edge of data eye not found.
51	stat(3)	PBD maximum reached in one or more channels.
52	stat(4)	Failed to find flag 0.
53	stat(5)	Failed to find flag 1.
54	stat(6)	Failed to find flag 2.
55	stat(7)	Final clock delay does not find IAP pattern.
56	stat(8)	Negative I/O clock delay.
57	riap_done	Receiver IAP completed.
58	ltc_passed	Learned target cycle passed.
59	ltc_failed	Learned target cycle failed.
60	est/rdt_passed	EST or RDT passed.

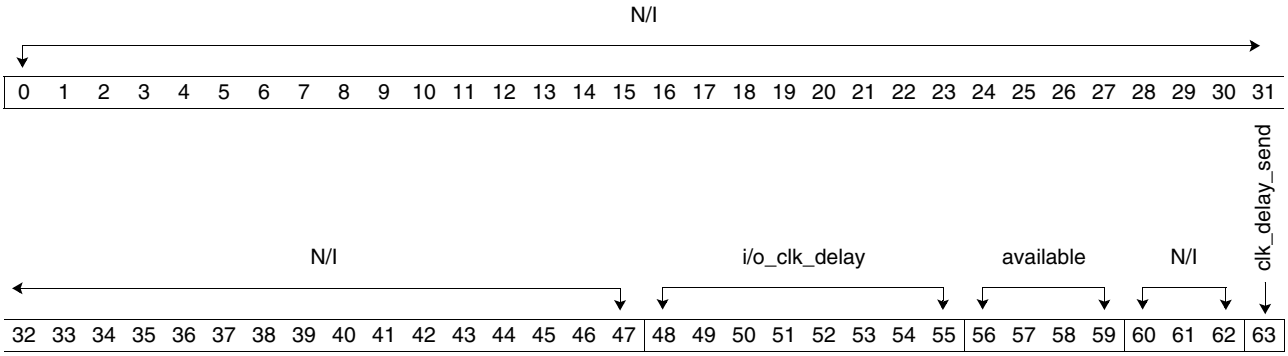
Bits	Field Name	Description
61	est/rdt_failed	EST or RDT failed.
62:63	available	Not used but available.

PI Command Register

Address x'085000'

Type WO

Reset Reset to all zeros during POR.



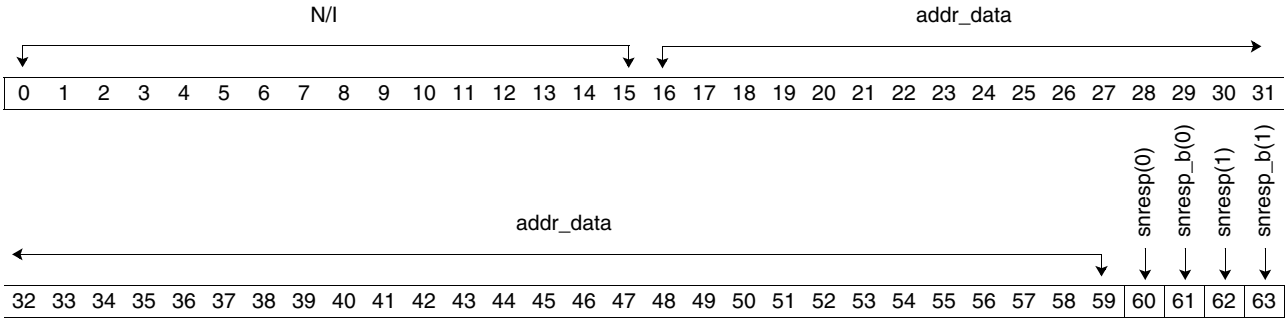
Bits	Field Name	Description
0:47	N/I	Not implemented.
48:55	i/o_clk_delay	Value of I/O clock delay to be loaded.
56:59	available	Not used but available.
60:62	N/I	Not implemented.
63	clk_delay_send	Set this bit to send the I/O clock delay to the I/O control macro.

Driver IAP Register
Receiver IAP Register

Address x'086000' (Driver)
x'086101' (Receiver)

Type R/W

Reset Reset to all zeros during POR.



Bits	Field Name	Description
0:15	N/I	Not implemented.
16:59	addr_data	Mask for adin/out(0:43).
60	snresp(0)	Mask for srin/out(0).
61	snresp_b(0)	Mask for srin/out_b(0).
62	snresp(1)	Mask for srin/out(1).
63	snresp_b(1)	Mask for srin/out_b(1).
Note: To take effect, requires est_one set (PI Mode Register 0, bit 52).		

12.5 Chip Pervasive SCOM Register Definition

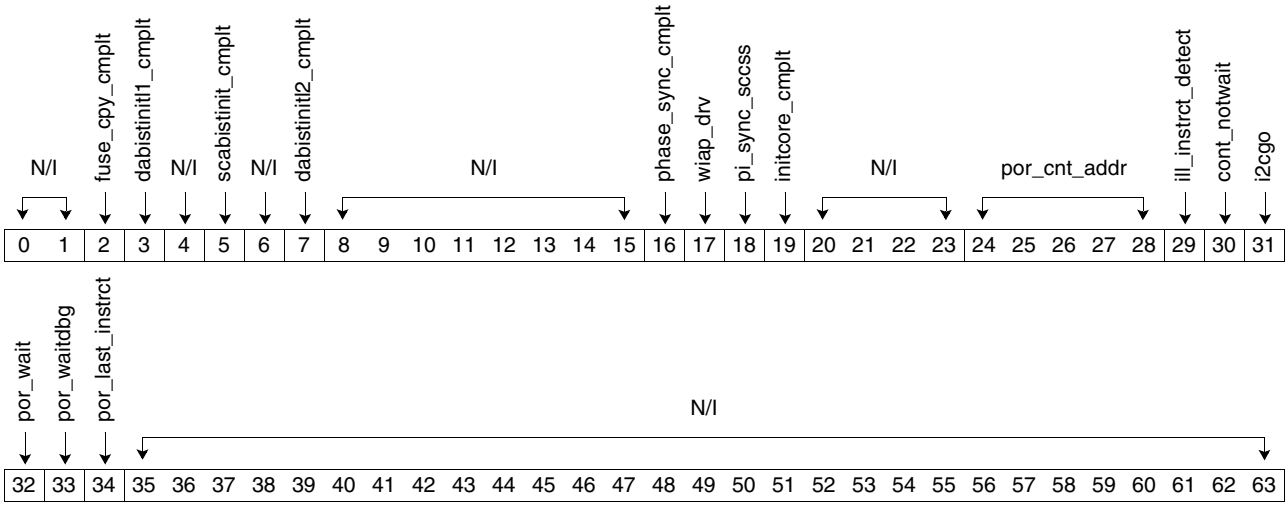
12.5.1 Power-On Reset Registers (x'40XXXX')

Power-On Reset Status Register

Address x'400000'

Type RO/WO
Clear entries are marked with an asterisk (*).

Reset Reset to all zeros during POR (inactive).



Bits	Field Name	Clear Entry	Description
0:1	N/I		Not implemented.
2	fuse_cpy_cmplt	*	Status is active after fuse copy completion.
3	dabistinitl1_cmplt	*	Status is active after DABISTINITL1 completion.
4	N/I		Not implemented.
5	scabistinit_cmplt	*	Status is active after SCABISTINIT completion.
6	N/I		Not implemented.
7	dabistinitl2_cmplt	*	Status is active after DABISTINITL2 completion.
8:15	N/I		Not implemented.
16	phase_sync_cmplt	*	Status is active after phase synchronization completion.
17	wiap_drv		Status is active while the writer initial alignment pattern (WIAP) is driven by POR.
18	pi_sync_sccss	*	Status is active after successful processor interface synchronization.
19	initcore_cmplt	*	Status is active after INITCORE completion.
20:23	N/I		Not implemented.
24:28	por_cnt_addr		Contains the current POR program counter address.

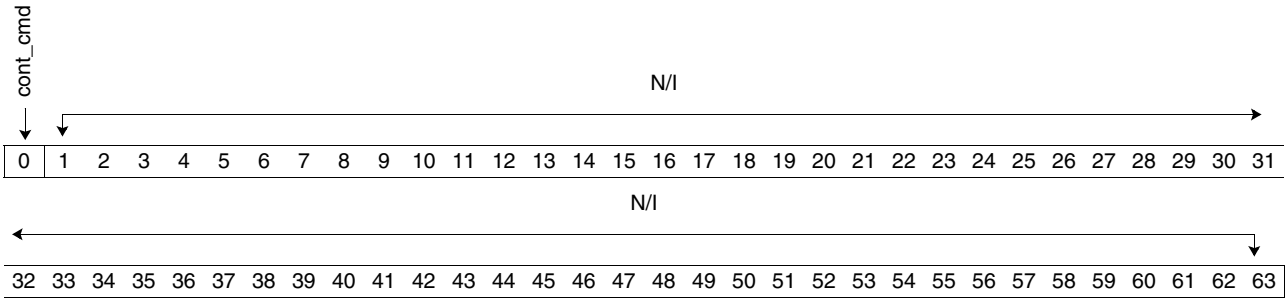
Bits	Field Name	Clear Entry	Description
29	ill_instrct_detect	*	Status is active after an illegal instruction is detected.
30	cont_notwait	*	Status is active if a continue was received while not in the Wait state.
31	i2cgo		Status is active after an I2CGO command until the next CONT command.
32	por_wait		Status is active when the POR state machine is in the Wait state.
33	por_waitdbg		Status is active when in the POR state machine is in the Waitdbg state.
34	por_last_instrct		Status is active when the POR state machine has reached the last instruction.
35:63	N/I		Not implemented.

Power-On Reset Continue Register

Address x'400101'

Type WO

Reset Reset to inactive during POR.



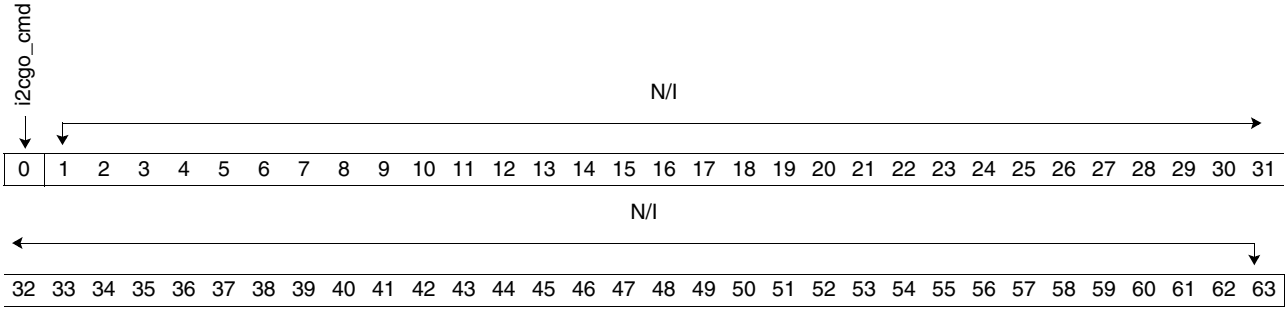
Bits	Field Name	Description
0	cont_cmd	Command bit. Sends a continue command to the POR state machine. This command also invalidates the *i2cgo* pin the next time the JTAG state machine enters the Test-Mode-Reset state or the Run-Idle state.
1:63	N/I	Not implemented.

Power-On Reset I^2C/JTAG Arbitration Register

Address x'400201'

Type WO

Reset Reset to inactive during POR.



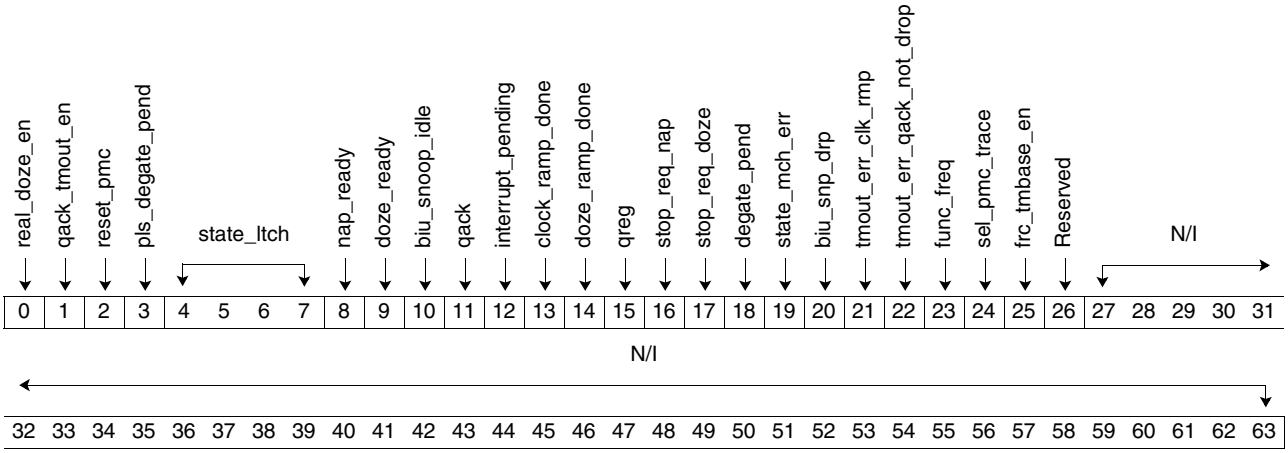
Bits	Field Name	Description
0	i2cgo_cmd	Command bit. Asserts and holds the *i2cgo* pin when the JTAG state machine enters the Test-Mode-Reset or Run-Idle state.
1:63	N/I	Not implemented.

Power-Management Control

Address x'400801'

Type Bits 0:3 and 24:26: RW
Bits 4:22: RO

Reset Reset to all zeros during POR.



Bits	Field Name	Description
0	real_doze_en	Real Doze mode enable.
1	qack_tmout_en	Advance on quiescent acknowledgment (QACK) drop timeout enable.
2	reset_pmc	Reset power management control state machine.
3	pls_degate_pend	Pulse degate pending.
PMC Status Bits		
4:7	state_ltch	State latches.
8	nap_ready	Nap_ready received.
9	doze_ready	Doze_ready received.
10	biu_snoop_idle	biu_snoop_idle received.
11	qack	QACK received.
12	interrupt_pending	interrupt_pending received.
13	clock_ramp_done	clock_ramp_done received.
14	doze_ramp_done	doze_ramp_done received.
15	qreg	QREG sent.
16	stop_req_nap	stop_req_nap sent.
17	stop_req_doze	stop_req_doze sent.
18	degate_pend	degate_pending sent.

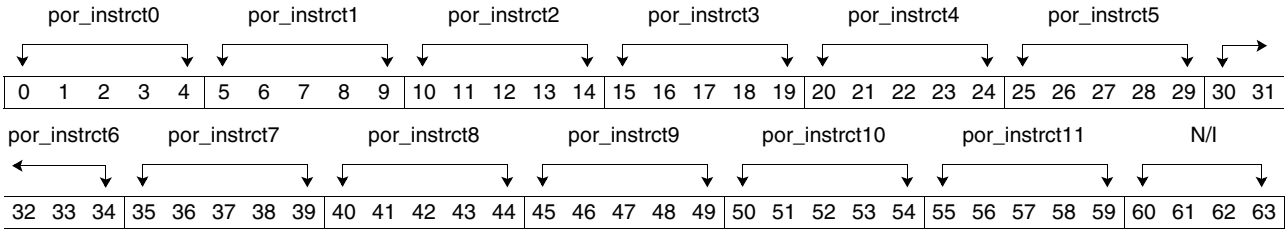
Bits	Field Name	Description
PMC Error Bits		
19	state_mch_err	State-machine error (undefined state).
20	biu_snp_drp	BIU Snoop Idle dropped while QACK active.
21	tmout_err_clk_rmp	Timeout error on clock ramp done.
22	tmout_err_qack_ not_drop	Timeout error on QACK not dropped.
Miscellaneous		
23	func_freq	If set, allows f/64 as the functional frequency. In addition, if this bit is set, it sets the write margin in dynamic arrays to the maximum for the f/2 and f/4 frequencies.
24	sel_pmc_trace	Select PMC tracing.
25	frc_tmbase_en	Force time-base enable.
26	Reserved	Reserved.
27:63	N/I	Not implemented.

Power-On Reset Sequence Register 0

Address x'401400'

Type WO

Reset Initialized during POR (see the *Power-On Reset Specification* for more information).



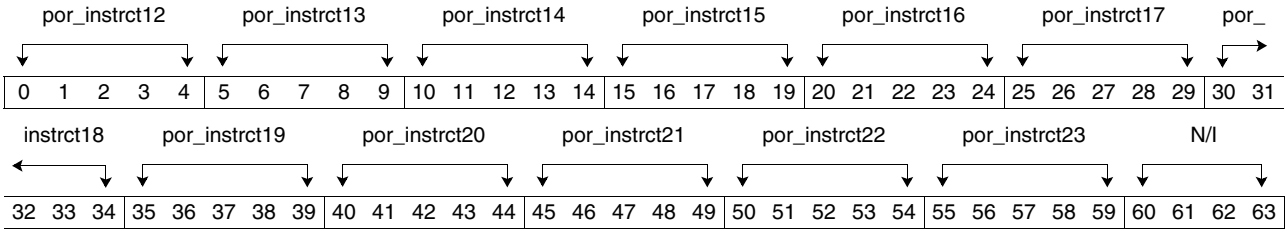
Bits	Field Name	Description
0:4	por_instrct0	Data: POR instruction 0 (first instruction).
5:9	por_instrct1	Data: POR instruction 1.
10:14	por_instrct2	Data: POR instruction 2.
15:19	por_instrct3	Data: POR instruction 3.
20:24	por_instrct4	Data: POR instruction 4.
25:29	por_instrct5	Data: POR instruction 5.
30:34	por_instrct6	Data: POR instruction 6.
35:39	por_instrct7	Data: POR instruction 7.
40:44	por_instrct8	Data: POR instruction 8.
45:49	por_instrct9	Data: POR instruction 9.
50:54	por_instrct10	Data: POR instruction 10.
55:59	por_instrct11	Data: POR instruction 11.
60:63	N/I	Not implemented.

Power-On Reset Sequence Register 1

Address x'402400'

Type WO

Reset Initialized during POR (see the *Power-On Reset Specification* for more information).



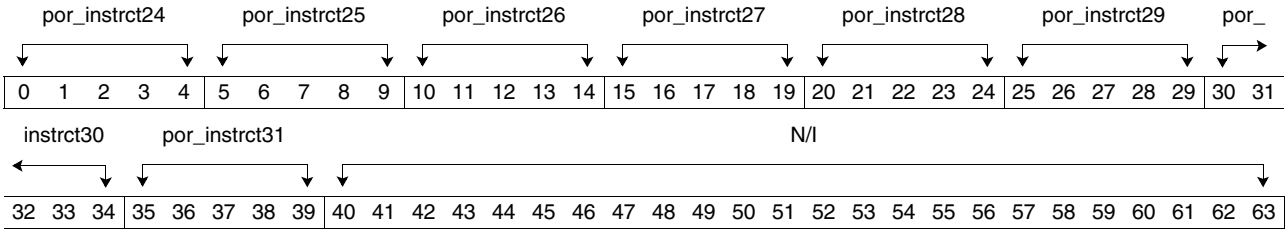
Bits	Field Name	Description
0:4	por_instrct12	Data: POR instruction 12.
5:9	por_instrct13	Data: POR instruction 13.
10:14	por_instrct14	Data: POR instruction 14.
15:19	por_instrct15	Data: POR instruction 15.
20:24	por_instrct16	Data: POR instruction 16.
25:29	por_instrct17	Data: POR instruction 17.
30:34	por_instrct18	Data: POR instruction 18.
35:39	por_instrct19	Data: POR instruction 19.
40:44	por_instrct20	Data: POR instruction 20.
45:49	por_instrct21	Data: POR instruction 21.
50:54	por_instrct22	Data: POR instruction 22.
55:59	por_instrct23	Data: POR instruction 23.
60:63	N/I	Not implemented

Power-On Reset Sequence Register 2

Address x'404400'

Type WO

Reset Initialized during POR (see the *Power-On Reset Specification* for more information).



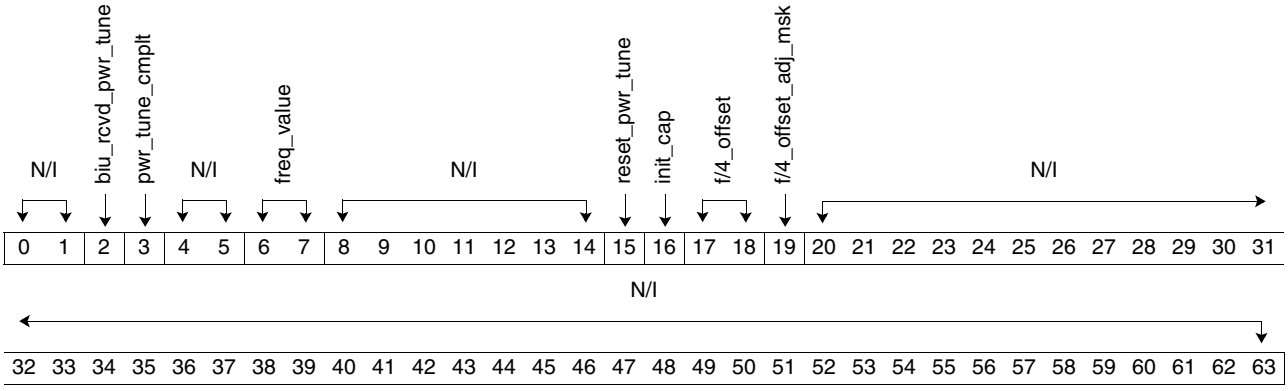
Bits	Field Name	Description
0:4	por_instrct24	Data: POR instruction 24.
5:9	por_instrct25	Data: POR instruction 25.
10:14	por_instrct26	Data: POR instruction 26.
15:19	por_instrct27	Data: POR instruction 27.
20:24	por_instrct28	Data: POR instruction 28.
25:29	por_instrct29	Data: POR instruction 29.
30:34	por_instrct30	Data: POR instruction 30.
35:39	por_instrct31	Data: POR instruction 31 (last instruction).
40:63	N/I	Not implemented.

Power Tuning Status Register

Address x'408001'

Type RW: bits 2:3, 7:8,16:19
WO: bit 15

Reset Initialized to all zeros during POR.



Bits	Field Name	Description
0:1	N/I	Not implemented.
2	biu_rcvd_pwr_tune	Turned on when the BIU has received a power tuning command from the North Bridge. Reset after a read to this register, and after power tuning command completion.
3	pwr_tune_cmplt	Turned on after completion of a power tuning command. Reset after a read to this register, and after power tuning command completion.
4:5	N/I	Not implemented.
6:7	freq_value	Current/requested frequency value.
8:14	N/I	Not implemented.
15	reset_pwr_tune	Reset the power tuning machine inside ChipRAS. The BIU received the normal power tuning complete acknowledgment. No error is reported.
16	init_cap	Initialize clock alignment procedure (CAP). Set and then unset to start CAP.
17:18	f/4_offset	Read: Computed the low-speed (f/4) frequency offset to SYSCLK. Write: Value to add to the f/4-frequency offset counter. Then toggle bit 19 to execute.
19	f/4_offset_adj_msk	Mask the f/4 offset counter adjust command. Set only to measure offset.
20:63	N/I	Not implemented.

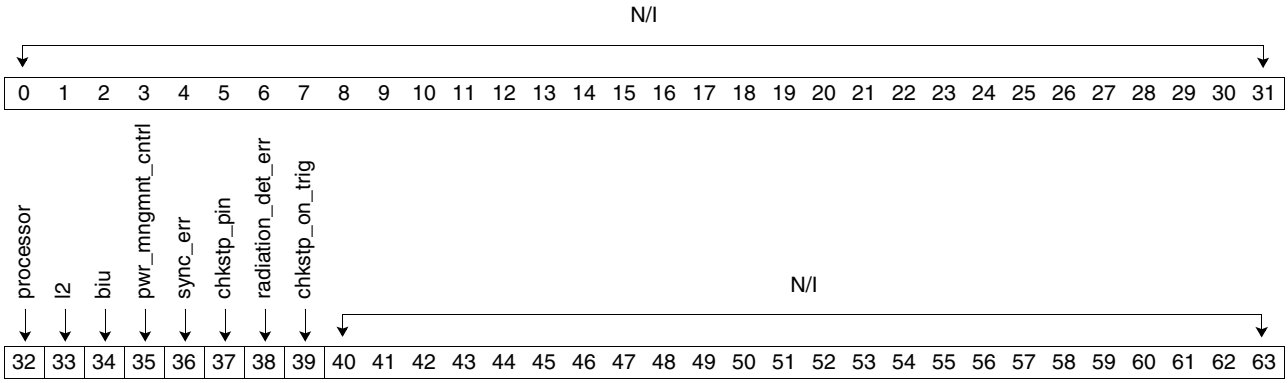
12.5.2 Chip Free-Running Clock Section Control/Status (x'50[0:4]XXX')

Global Fault Isolation for Checkstop Conditions (Global FIR)

Address x'500001'

Type RO

Reset Reset to all zeros during POR.



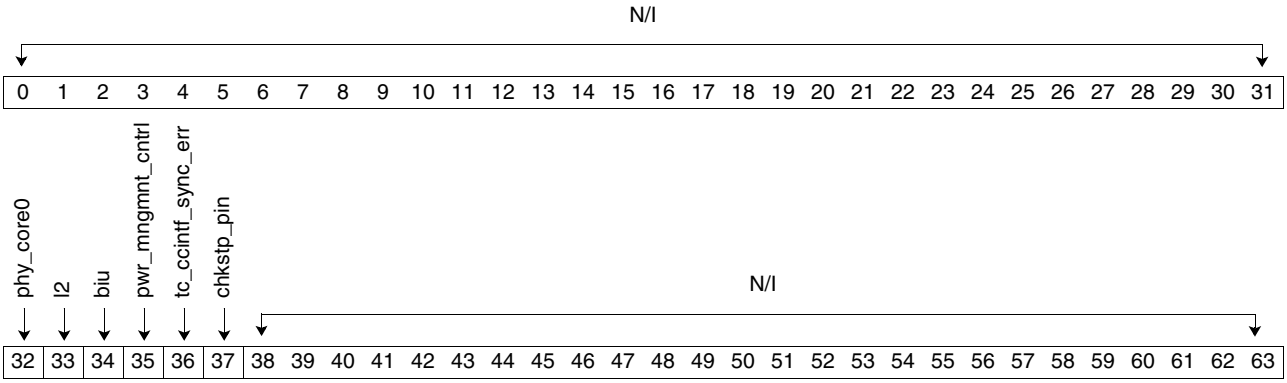
Bits	Field Name	Description
0:31	N/I	Not implemented.
32	processor	Processor core.
33	l2	L2.
34	biu	BIU.
35	pwr_mngmnt_cntrl	Power-management control.
36	sync_err	Synchronization error in the free-running clock controls macro, tc_ccintf.
37	chkstp_pin	Checkstop pin.
38	radiation_det_err	Radiation detection error.
39	chkstp_on_trig	Checkstop on trigger.
40:63	N/I	Not implemented.

Error Enable Mask

Address x'500400'

Type RW

Reset Reset to all zeros during POR.



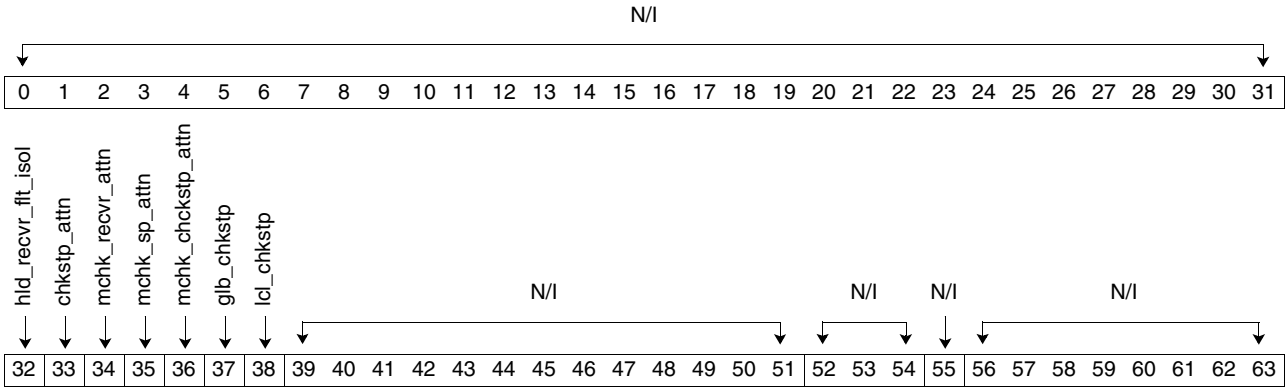
Bits	Field Name	Description
0:31	N/I	Not implemented.
32	phy_core0	Physical core0.
33	l2	L2.
34	biu	BIU.
35	pwr_mngmnt_cntrl	Power-management control.
36	tc_ccintf_sync_err	Synchronization error in TC_CCINTF.
37	chkstp_pin	Checkstop pin.
38:63	N/I	Not implemented.

Mode Register for Fault Isolation Registers

Address x'500601'

Type RW

Reset Reset to all zeros during POR.



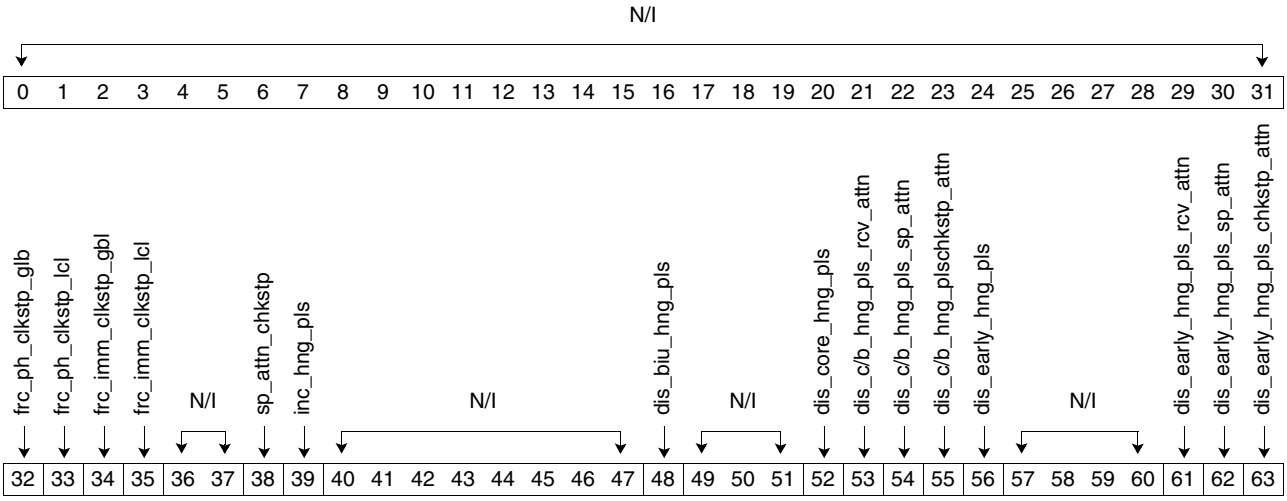
Bits	Field Name	Description
0:31	N/I	Not implemented.
32	hld_recvr_flt_isol	Hold recoverable-fault isolation when error is detected. Behaves like the checkstop FIR.
33	chkstp_attn	Checkstop indications from the checkstop pin, or checkstop trigger will set the checkstop attention.
34	mchk_recvr_attn	Processor machine check will set recoverable attention.
35	mchk_sp_attn	Processor machine check will set special attention.
36	mchk_chckstp_ attn	Processor machine check will set checkstop attention.
37	glb_chkstp	Global checkstop signal will signal checkstop to processor cores.
38	lcl_chkstp	Local checkstop signal will signal checkstop to processor cores.
39:63	N/I	Not implemented.

Debug Mode Register

Address x'500700'

Type RW

Reset Reset to all zeros during POR.



Bits	Field Name	Description
0:31	N/I	Not implemented.
32	frc_ph_clkstp_glb	Force a phase-aligned clock stop when the global checkstop signal is active.
33	frc_ph_clkstp_lcl	Force a phase-aligned clock stop when the local checkstop signal is active.
34	frc_imm_clkstp_ gbl	Force an immediate clock stop when the global checkstop signal is active.
35	frc_imm_clkstp_lcl	Force an immediate clock stop when the local checkstop signal is active.
36:37	N/I	Not implemented.
38	sp_attn_chkstp	Core special attention will cause a checkstop.
39	inc_hng_pls	Increase hang pulse rate by 100 times.
40:47	N/I	Not implemented.
48	dis_biu_hng_pls	Disable BIU hang pulses.
49:51	N/I	Not implemented.
52	dis_core_hng_pls	Disable core hang pulses.
53	dis_c/b_hng_pls _rcv_attn	Disable core and BIU hang pulses when recoverable attention is set.
54	dis_c/b_hng_pls _sp_attn	Disable core and BIU hang pulses when special attention is set.
55	dis_c/b_hng_pls chkstp_attn	Disable core and BIU hang pulses when checkstop attention is set.
56	dis_early_hng_pls	Disable early hang pulse to core.
57:60	N/I	Not implemented.

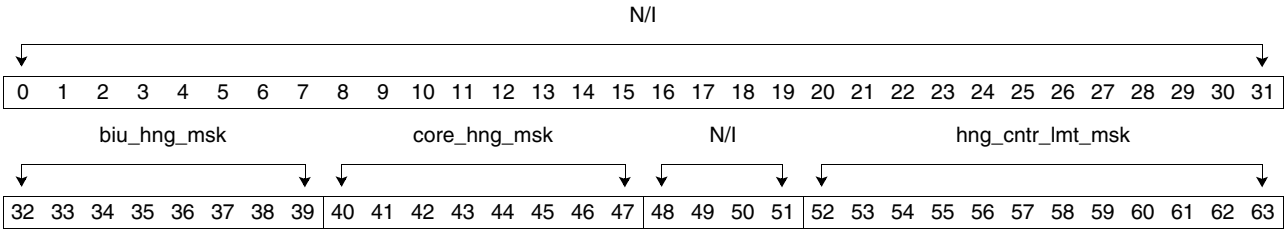
Bits	Field Name	Description
61	dis_early_hng_pls _rcv_attn	Disable early hang pulses when recoverable attention is set.
62	dis_early_hng_pls _sp_attn	Disable early hang pulses when special attention is set.
63	dis_early_hng_pls _chkstp_attn	Disable early hang pulses when checkstop attention is set.

Hang Pulse Generation

Address x'503001'

Type RW

Reset Reset to all zeros during POR.



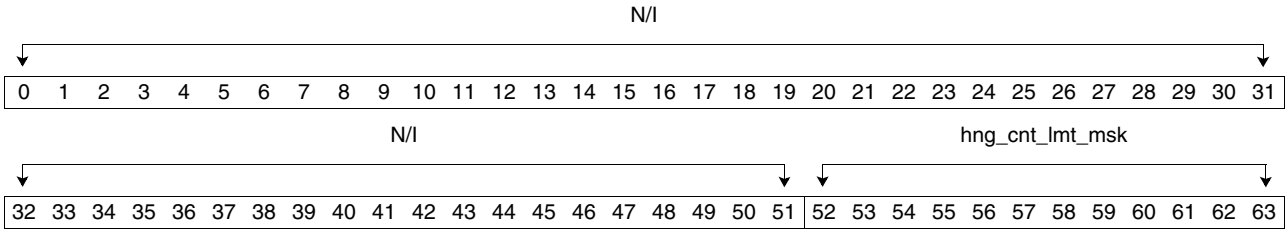
Bits	Field Name	Description
0:31	N/I	Not implemented.
32:39	biu_hng_msk	BIU hang mask. x'01' Hang pulse generated every *Counter x 2.* x'02' Hang pulse generated every *Counter x 4.* x'04' Hang pulse generated every *Counter x 8.* x'08' Hang pulse generated every *Counter x 16.* x'10' Hang pulse generated every *Counter x 32.* x'20' Hang pulse generated every *Counter x 64.* x'40' Hang pulse generated every *Counter x 128.* x'80' Hang pulse generated every *Counter x 256.*
40:47	core_hng_msk	Core hang mask. x'01' Hang pulse generated every *Counter x 2.* x'02' Hang pulse generated every *Counter x 4.* x'04' Hang pulse generated every *Counter x 8.* x'08' Hang pulse generated every *Counter x 16.* x'10' Hang pulse generated every *Counter x 32.* x'20' Hang pulse generated every *Counter x 64.* x'40' Hang pulse generated every *Counter x 128.* x'80' Hang pulse generated every *Counter x 256.*
48:51	N/I	Not implemented.
52:63	hng_cntr_lmt_msk	Hang counter limit mask (12-bit LFSR). x'4FC' 100 cycles x'CF0' 200 cycles x'DAE' 300 cycles x'E89' 400 cycles x'D1A' 500 cycles x'0A3' 600 cycles x'F8C' 700 cycles x'FAA' 800 cycles x'8F3' 900 cycles x'6C8' 1000 cycles x'5D1' 2000 cycles x'668' 3000 cycles x'D3E' 4000 cycles

Early Hang Pulse Generation

Address x'503100'

Type RW

Reset Reset to all zeros during POR.



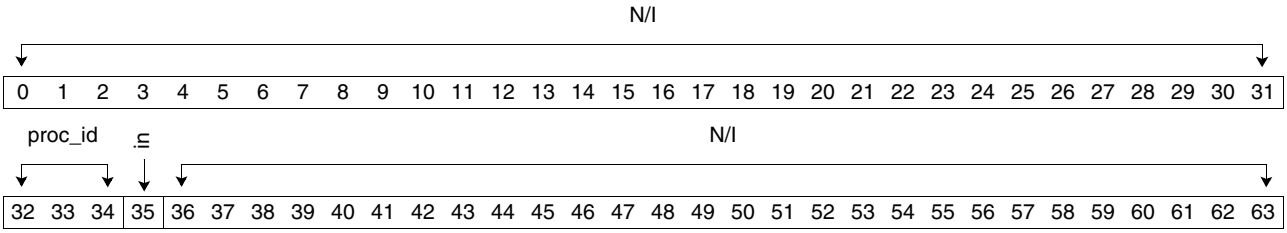
Bits	Field Name	Description
0:51	N/I	Not implemented.
52:63	hng_cnt_lmt_msk	Hang counter limit mask (12-bit LFSR). x'6A8' 20 cycles x'BA0' 50 cycles x'4FC' 100 cycles x'CF0' 200 cycles x'DAE' 300 cycles x'E89' 400 cycles x'D1A' 500 cycles x'0A3' 600 cycles x'F8C' 700 cycles x'FAA' 800 cycles x'8F3' 900 cycles x'6C8' 1000 cycles x'5D1' 2000 cycles x'668' 3000 cycles x'D3E' 4000 cycles

Chip ID Register

Address x'504101'

Type RW (Bit 35 is WO)

Reset Reset to zeros during POR.



Bits	Field Name	Description
0:31	N/I	Not implemented.
32:34	proc_id	Processor ID (0:2).
35	in	Capture values from processor ID (PID) primary C4 inputs. (This is a write only bit.)
36:63	N/I	Not implemented. Returns zeros.

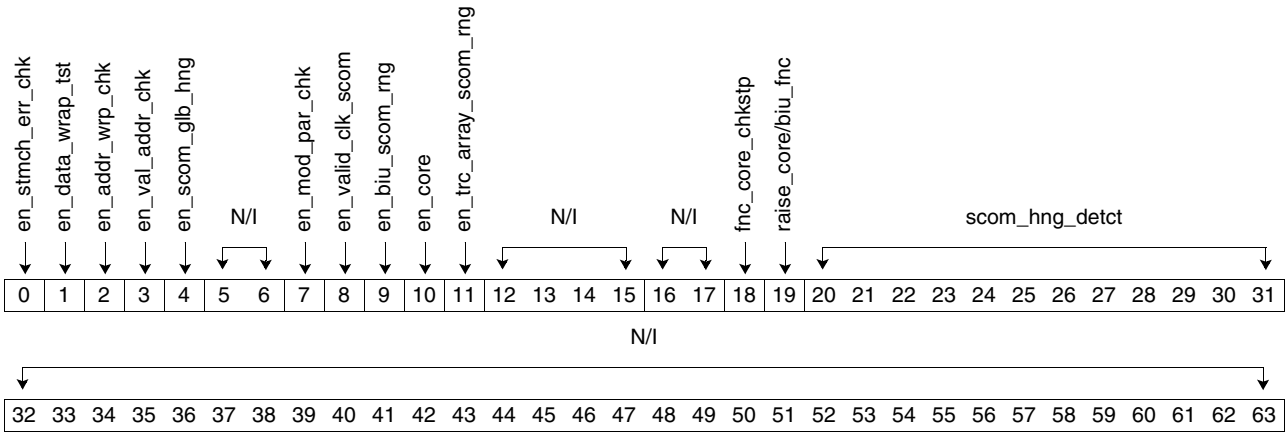
12.5.3 Chip Parallel SCOM Control (x'6XXXXX')

SCOM Mode Register

Address x'600001'

Type R/W

Reset Set to x'0000 0FFE 0000 0000' during POR or SCRESET.



Bits	Field Name	Description
0	en_stmch_err_chk	Enable state-machine error checking.
1	en_data_wrap_tst	Enable data wrap test.
2	en_addr_wrp_chk	Enable address wrap check.
3	en_val_addr_chk	Enable valid address checking.
4	en_scom_glb_hng	Enable SCOM global hang checking.
5:6	N/I	Not implemented.
7	en_mod_par_chk	Enable modifier parity checking.
8	en_valid_clk_scom	Enable valid clock SCOM address checking.
9	en_biu_scom_rng	Enable STS/BIU SCOM ring. Set to '0' before scanning BIU.
10	en_core	Enable core SCOM ring. Enables SCOM ring and functional fences. Set to '0' before scanning core.
11	en_trc_array_ scom_rng	Enable I/O SCOM ring. Set to '0' before scanning I/O.
12:17	N/I	Not implemented.
18	fnc_core_chkstp	Fence cores on checkstop. **Note:** Bits 37:38 of the Mode Register for Fault Isolation Registers controls the core checkstop.
19	raise_core/biu_fnc	Raise core and BIU fences on a clock stop. **Note:** A clock stop is seen by the free-running logic several cycles before the clocks are actually stopped. By default, software should set this bit to '1'. However, it must be set to '0' before cycle stepping for debug.

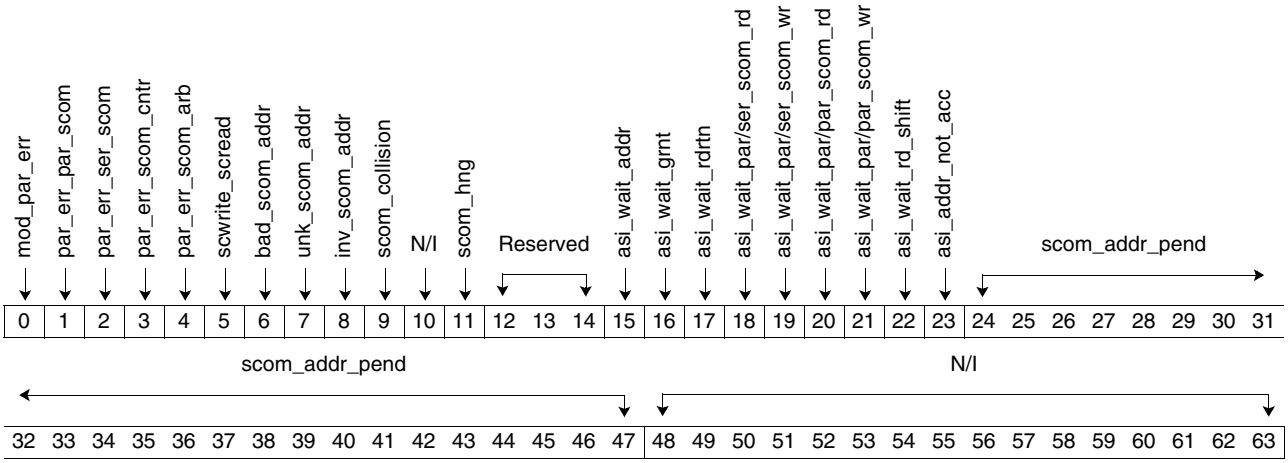
Bits	Field Name	Description
20:31	scom_hng_detct	When the LFSR counter matches the value in the register before the SCOM operation completes, a SCOM hang has been detected. Processor Clock Cycles: / LFSR Counter Values: 500 / x'D1A' 1000 / x'6C8' 2000 / x'5D1' 3000 / x'668' 4095 / x'FFE'
32:63	N/I	Not implemented.

SCOM Controller Error Register

Address x'600100'

Type RO (Write zeros to clear after SCRESET).

Reset Reset to all zeros during POR.



Bits	Field Name	Description
0	mod_par_err	Modifier parity error.
1	par_err_par_scom	Parity error in the parallel SCOM state machine.
2	par_err_ser_scom	Parity error in the serial SCOM state machine.
3	par_err_scom_cntr	Parity error in the main SCOM controller state machine.
4	par_err_scom_arb	Parity error in the SCOM arbiter state machine.
5	scwrite_scread	The SCWRITE pipe latch and SCREAD pipe latch were both on.
6	bad_scom_addr	Bad address in the SCOM ring.
7	unk_scom_addr	The SCOM address was not recognized.
8	inv_scom_addr	The SCOM address for the clock command was invalid.
9	scom_collision	An SCOM collision in the core.
10	N/I	Not implemented.
11	scom_hng	SCOM hang. The active source identifier (ASI) in bits 15:23 indicates what was pending.
12:14	Reserved	Reserved.
15	asi_wait_addr	(ASI) Waiting for the address to return.
16	asi_wait_grnt	(ASI) Waiting for a grant from the arbiter.
17	asi_wait_rdrtn	(ASI) Waiting for read data to return.
18	asi_wait_par/ser_scom_rd	(ASI) The parallel-to-serial state machine is waiting for SCOM Read to drop.
19	asi_wait_par/ser_scom_wr	(ASI) The parallel-to-serial state machine is waiting for SCOM Write to drop.
20	asi_wait_par/par_scom_rd	(ASI) The parallel-to-parallel state machine is waiting for SCOM Read to drop.

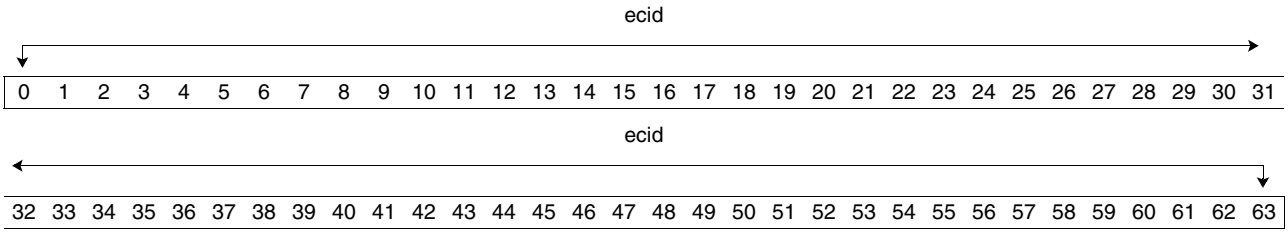
Bits	Field Name	Description
21	asi_wait_par/par_scom_wr	(ASI) The parallel-to-parallel state machine is waiting for SCOM Write to drop.
22	asi_wait_rd_shift	(ASI) Waiting for the read shifter to empty.
23	asi_addr_not_acc	(ASI) The address was returned and not accepted.
24:47	scom_addr_pend	The SCOM address that was pending at the time of the error.
48:63	N/I	Not implemented.

Electronic Chip ID

Address x'600200'

Type RO

Reset Set to fuse values when the fuse ring is scanned.



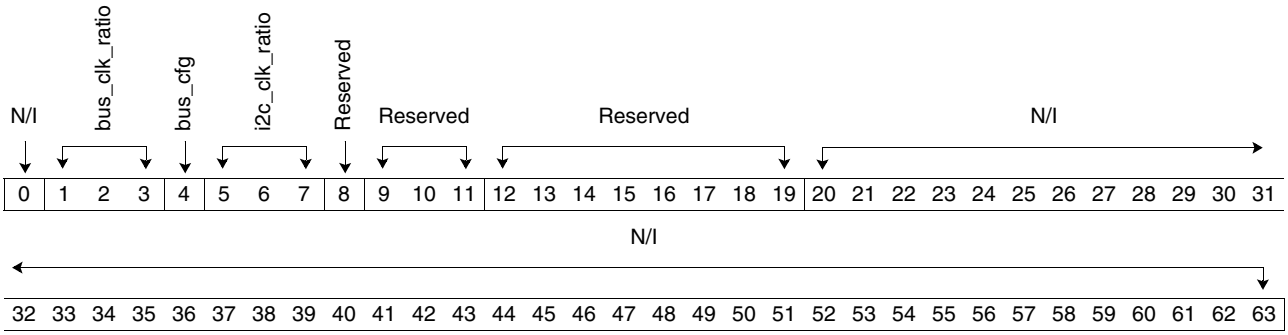
Bits	Field Name	Description
0:63	ecid	Electronic chip ID (ECID).

Clock Ratio Register (N:1 Phase Hold Control)

Address x'600400'

Type RW

Reset ratio: Reset to all zeros during POR.
iap_encode: Reset to '1' during POR.
count1us_factor: Reset to all ones during POR.



Bits	Field Name	Description
0	N/I	Not implemented.
1:3	bus_clk_ratio	Bus clock ratio (for N:1 phase_hold generation). 000 2:1 001 3:1 010 4:1 011 6:1 100 8:1 101 12:1 110 24:1 111 1:1
4	bus_cfg	Writing a '1' to this bit triggers the bus_cfg read operation, which captures values from the bus_cfg primary C4 inputs. Reads return '0'.
5:7	i2c_clk_ratio	I^2C clock ratio. This counter is asynchronous to the mod48 counter and scales with the processing unit frequency. 000 16:1 (full speed), 8:1 (half speed), 4:1 (quarter speed) 001 Not supported 010 8:1 (full speed), 4:1 (half speed), 2:1 (quarter speed) 011 Not supported 100 Not supported 101 Not supported 110 Not supported 111 1:1
8:19	Reserved	Reserved.
20:63	N/I	Not implemented.

12.5.4 Chip Clock/Scan Control (x'8[0:4]XXXX')

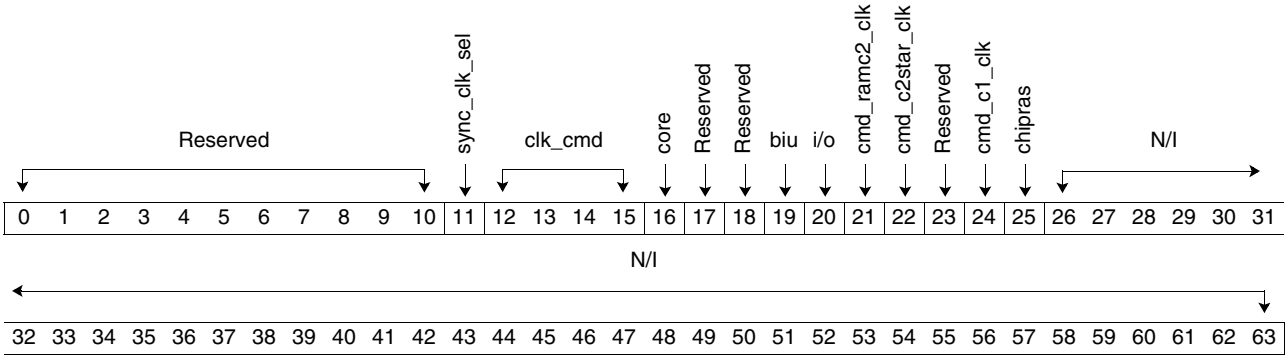
Clock Command Register

See *Figure 12-6 Common Clock Commands* on page 471.

Address x'800000'

Type R/W

Reset Set to all zeros during POR.



Bits	Field Name	Description
0:10	Reserved	Unused.
11	sync_clk_sel	SYNC clock select. Must be programmed to logic level zero ['0'].
12:15	clk_cmd	Clock command x'4' Pulse selected clocks x'8' Start selected clocks x'C' Stop selected clocks **Note:** Bits 14:15 are not implemented.
Domain Select		
16	core	Domain select core.
17:18	Reserved	Unused.
19	biu	Domain select STS/BIU.
20	i/o	Domain select I/O.
Array Clock Commands (applies to Start, Pulse, and Stop)		
21	cmd_ramc2_clk	Apply command to RAMC2 clock signal (not valid for I/O).
22	cmd_c2star_clk	Apply command to C2star clock signals.
23	Reserved	Unused.
Latch Clock Commands (applies to Start or Pulse, and Stop)		
24	cmd_c1_clk	Apply command to C1 clock signal.
Free-Running Clock Section		
25	chipras	Domain select: ChipRAS.
26:63	N/I	Not implemented.

Figure 12-6. Common Clock Commands

-- Start --
x'00088**68**000000000' Core

x'00081**68**000000000' BIU
x'00080**E8**000000000' I/O
x'0008**9E8**000000000' All

-- Stop --
x'000C8**68**000000000' Core

x'000C1**68**000000000' BIU
x'000C0**E8**000000000' I/O
x'000C**9E8**000000000' All
-- Clock 1 (C1/C2*/RAMC2) --
x'00048**68**000000000' Core

x'00041**68**000000000' BIU
x'00040**E8**000000000' I/O
x'0004**9E8**000000000' All

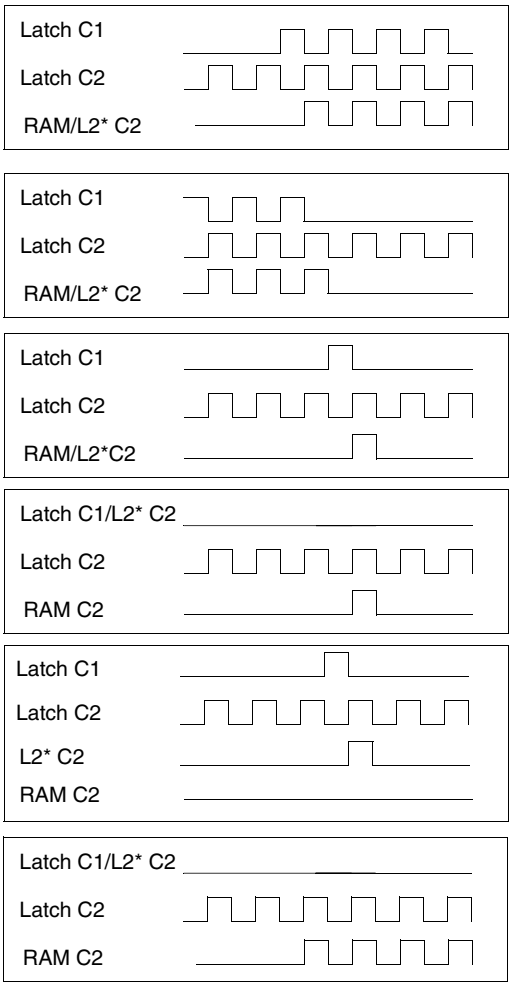
-- RAMC2 (RAM Clock only) --
x'0004840000000000' Core RAMC2

x'0004140000000000' BIU RAMC2

-- C1 (plus C2*) --
x'00048**28**000000000' Core C1
x
x'00041**28**000000000' BIU C1
x'00040**A8**000000000' I/O C1
x'0004**9A8**000000000' All C1

-- Start RAM Clock only --
x'0008840000000000' Core RAMC2

x'0008140000000000' BIU RAMC2
x'0008**9**40000000000' All RAMC2



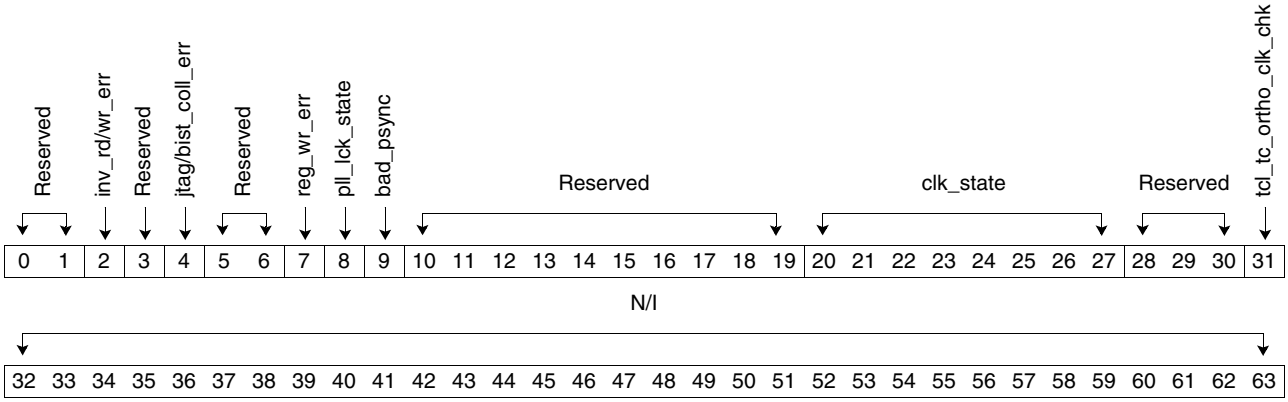
Status Register

Note: Bits 0:12 of the Status Register can be masked or blocked by setting bits 0:12 of the Status Register Mask.

Address x'800003'

Type R/W: bits 2, 4, 7:9, 31. RO: bits 20:21, 23:24

Reset Set to x'0000 0040 0000 0000' during POR.



Bits	Field Name	Description
0:1	Reserved	Reserved.
2	inv_rd/wr_err	Invalid read/write address error. An attempt was made to read or write to a clock interface address that does not exist. Raises SCATTN unless blocked.
3	Reserved	Reserved.
4	jtag/bist_coll_err	JTAG/BIST collision error. JTAG attempted to initiate a write to a valid register while the event processor or array built-in self test (ABIST) was in progress. Raises SCATTN unless blocked.
5:6	Reserved	Reserved.
7	reg_wr_err	Register write error. Either of the following actions raises SCATTN unless it is blocked: • A write to the Clock Command Register (address x'800009') while not all of the core0 or STS clocks are off. • A write to the I/O Control Register (address x'80000F') while the I/O clocks are running.
8	pll_lck_state	pll_lock state. If '1', PLL unlock is detected. Raises SCATTN unless blocked.
9	bad_psync	Bad *psync*[1] detected. The Clock Controls mod48 counter does not match the Power Tuning mod48 counter.
10:19	Reserved	Reserved.

1. A signal provided by the North Bridge, which is active for one rising edge of SYSCLK every 24 SYSCLK cycles.

Bits	Field Name	Description
20:27	clk_state	Clock state bits (CLK_STATE[0:7]). These correspond one-to-one with the domains defined in bits 16 to 20 of the Clock Command Register. A value of '1' indicates that the C! clocks are running. These bits cannot be written. Bits 20 Core 21 Pervasive 22 Reserved 23 STS 24 I/O 25:27 Reserved
28:30	Reserved	Reserved.
31	tcl_tc_ortho_clk_chk	Indicates that an attempt was made to scan a domain while the clocks were not stopped.
32:63	N/I	Not implemented.

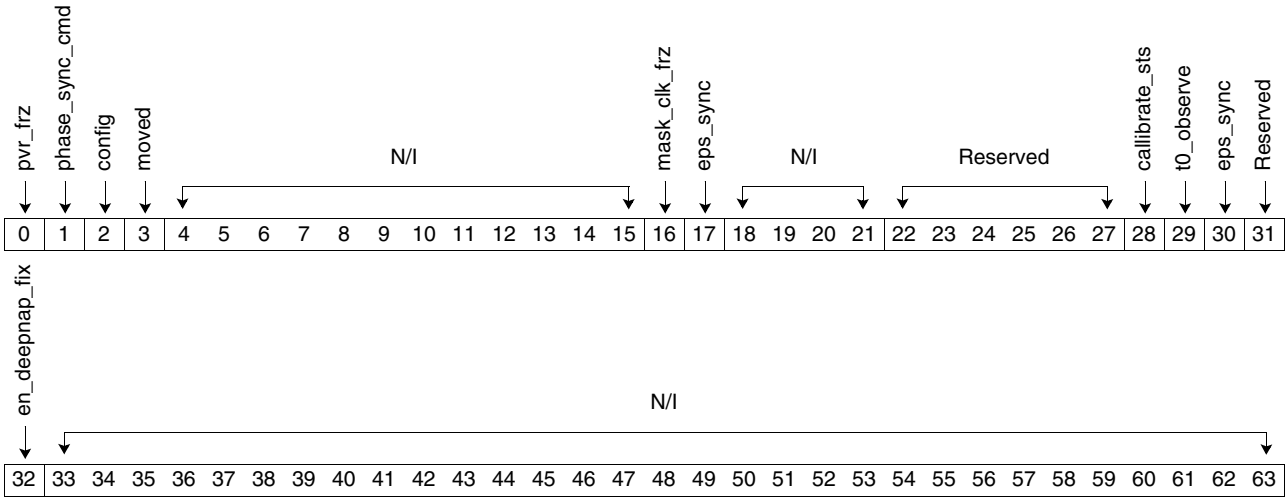
1. A signal provided by the North Bridge, which is active for one rising edge of SYSCLK every 24 SYSCLK cycles.

Phase Synchronization Control Register

Address x'800006'

Type R/W: 0:2, 16:17
WO: 22:27

Reset Set to all zeros during POR.



Bits	Field Name	Description
0	pvr_frz	Processor Version Register (PVR) freeze. If set, the PVR latches in the fixed-point unit (FXU) can be scanned to an arbitrary value.
1	phase_sync_cmd	Command bit. Run phase synchronization.
2	config	Configuration bit. Do not checkstop on bad_sync.
3	moved	Moved to bit 9 of the Status Register.
4:15	N/I	Not implemented.
16	mask_clk_frz	Mask clock freeze. When set to '1', the clock-freeze signal from the free-running global controls macro, ts_glob, is ignored.
17	eps_sync	Event processing sequencer (EPS) synchronization. If set, starts the EPS engine processing synchronously to the EPS engine of the other processing unit.
18:21	N/I	Not implemented.
22:27	Reserved	Reserved (all zeros).
28	callibrate_sts	STS calibrate. Used during test to generate a test-only (TO) initialization pulse to the STS.
29	t0_observe	T0 observation. If set, toggles the quiescent request (QREQ) pin on every T0 pulse.
30:31	Reserved	Reserved(all zeros).
32	en_deepnap_fix	Enable deep nap delaying to prevent clock race condition at certain bus ratios.
33:63	N/I	Not implemented.

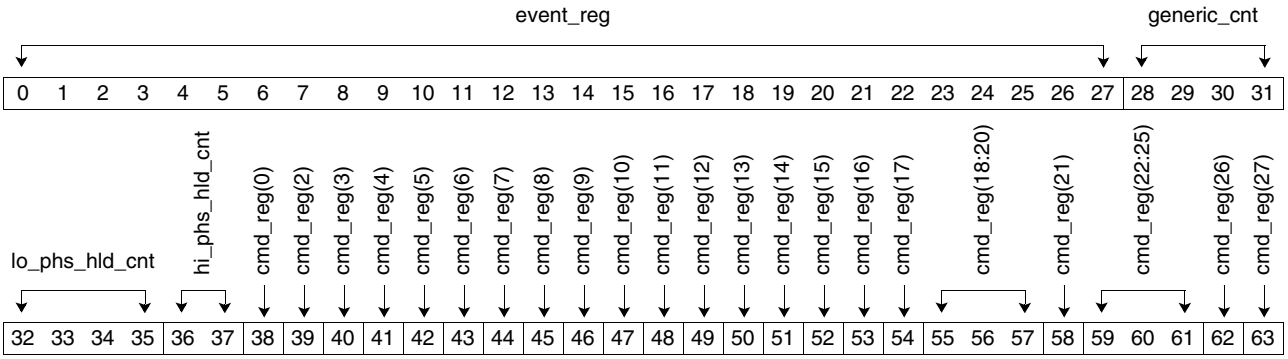
Clock Command Control Register

This SCOM register controls the Event Processing Sequencer (EPS) used to generate clock patterns for LBIST, ABIST, POR, and debugging.

Address x'800009'

Type R/W (Writing this register while the core or BIU is running will raise an attention.)

Reset Set to x'0000000 0314001A' during POR.



Bits	Field Name	Description
0:27	event_reg	Event register (four fields of 7 control bits per field). Bits 21:27 First event field. 14:20 Second event field. 7:13 Third event field. 0:6 Fourth event field. Field bit definitions: Bits 0, 7, 14, 21: Run system C1 clock. 1, 8, 15, 22: Run scan SC1 clock. 2, 9, 16, 23: Run RAM_C2 clock. 3, 10, 17, 24: Run C2STAR clock. 4, 11, 18, 25: Ignored. 5, 12, 19, 26: Enable clock-rate divide counter. This slows down the application of clock events. 6, 13, 20, 27: Enable generic counter, specifying the number of times to loop on the event field.
28:31	generic_cnt	Generic counter (0:3). Programmed count value: down counter.
32:35	lo_phs_hld_cnt	Lower-order phase hold counter (mod48) for event processing. Valid values are 0 to 11. **Note:** This counter cannot be used while the PHASESYNC POR instruction is active.

Notes:

- The system C2 clock is always running in functional mode.
- The domain selects (bits 38:42) only control the scope of C1, RAM_C2, and C2STAR.
- Bit 52 initiates this sequence, which performs the events in order. Bits 55:57 control looping on these events. The SCOM LBIST Test Length Register must be used to configure the TEST LENGTH count.
- If scan SC1 is the *only* clock selected for an event, then the CHANNEL LENGTH count is used for that event. The SCOM LBIST Channel Length Register must be used to configure the CHANNEL LENGTH count. If SC1 is selected in addition to either a SYSTEM or RAM clock, then only a single clock is given.

Bits	Field Name	Description
36:37	hi_phs_hld_cnt	Upper-order phase hold counter for event processing. Programmed count value: up counters 0 to 3. **Notes:** • This counter cannot be used while the PHASESYNC POR instruction is active. • This counter counts the carry out of the lower-order phase hold counter (for event processing).
38	cmd_reg(0)	CORE 0 select.
39	cmd_reg(2)	BIU select.
40	cmd_reg(3)	I/O select.
41	cmd_reg(4)	Free-running select.
42	cmd_reg(5)	Domain select for hard stop. Free-running domain.
43	cmd_reg(6)	Global array inhibit. This signal gets ORed with *ABIST_EN* and *Local_Latch* OR *ESP_RAMSCAN* and NOT *Local_Lt* OR LSSD/GSD Scan-Active. If the scan 0 has not occurred through the scan-ABIST section, the arrays will be enabled functionally when the global array inhibit is inactive. Set during POR. Need to clear by writing.
44	cmd_reg(7)	SCAN 0 command (similar to the FLUSH 0 Access Command). Scan in '0' to all the selected domains specified by Cmd_reg(0:3) *except* the FUSE and NOT_BIST (timing chain) rings.
45	cmd_reg(8)	FULL SCAN 0 command. Used in conjunction with the SCAN 0 Command. Causes the FUSE and NOT_BIST (timing chain) to also scan to '0'. POR sets it to '1' (clean NOT_BIST chain).
46	cmd_reg(9)	LBIST. When set to run LBIST, causes LBIST ring-selects and causes the Multiple Input Signature Register (MISR) connections to be established. The LBIST logic is fenced out in the free-running domain. This bit has no effect on the EPS engine. Bit 11 in the I/O Control Register (x'80000F') is automatically set to fence the I^2C. It is not reset when the LBIST bit is reset.
47	cmd_reg(10)	CAM BIST. Set for testing content-addressable memory (CAM) arrays with scanned ABIST.
48	cmd_reg(11)	Central scanned ABIST core. Set to test core arrays covered by scanned ABIST.
49	cmd_reg(12)	Central scanned ABIST not core. Set to test noncore arrays covered by scanned ABIST.
50	cmd_reg(13)	Decentral ABIST. Set to test arrays covered by the following dedicated ABIST engines: L2 cache, L2 directory, L2 least recently used (LRU), IFU cache, IFU directory, IFU branch history table (BHT), L1 cache data (L1D).
51	cmd_reg(14)	Decentral ABIST RAM_SELECT. Zeros for the first half of the arrays; ones for the second half.
52	cmd_reg(15)	Event processing command. Initiates a clock event process (see *Figure 12-7* on page 479 for more information).
53	cmd_reg(16)	Phase load control(0). Load the phase counter with the programmed value at event processor state 2.
54	cmd_reg(17)	Phase load control(1). Load the phase counter with the programmed value with Event Complete. Normally, for LBIST, this bit would be set. Not programming this bit makes it possible to change the RUN_CLOCK pulse to PHASE_HOLD alignment.

Notes:

- The system C2 clock is always running in functional mode.
- The domain selects (bits 38:42) only control the scope of C1, RAM_C2, and C2STAR.
- Bit 52 initiates this sequence, which performs the events in order. Bits 55:57 control looping on these events. The SCOM LBIST Test Length Register must be used to configure the TEST LENGTH count.
- If scan SC1 is the *only* clock selected for an event, then the CHANNEL LENGTH count is used for that event. The SCOM LBIST Channel Length Register must be used to configure the CHANNEL LENGTH count. If SC1 is selected in addition to either a SYSTEM or RAM clock, then only a single clock is given.

Bits	Field Name	Description
55:57	cmd_reg(18:20)	Run_N Command control. Causes the event processor to loop back to different event fields. This provides the capability of an initialization setup. 000 Loop on all events. Branches back to first event. 100 Loop on fourth event. 101 Branch back to second event. 110 Branch back to third event.
58	cmd_reg(21)	Event processing override. Provides the means to interrupt an event process that is in progress.
59:61	cmd_reg(22:25)	Domain select for hard stop. SCOM bits / cmd_reg bit 59 / 22 Core active high. 23 Reserved. 60 / 24 STS active high. 61 / 25 I/O active high.
62	cmd_reg(26)	I/O mesh clock select. The processor interface runs off an *input* clock called the IO_CLOCK. For LBIST and scanning of the I/O domain, this bit should be set first. POR sets this bit to '1'.
63	cmd_reg(27)	LBIST ac mode. Causes 8:1 logic clocks to turn off.

Notes:

- The system C2 clock is always running in functional mode.
- The domain selects (bits 38:42) only control the scope of C1, RAM_C2, and C2STAR.
- Bit 52 initiates this sequence, which performs the events in order. Bits 55:57 control looping on these events. The SCOM LBIST Test Length Register must be used to configure the TEST LENGTH count.
- If scan SC1 is the *only* clock selected for an event, then the CHANNEL LENGTH count is used for that event. The SCOM LBIST Channel Length Register must be used to configure the CHANNEL LENGTH count. If SC1 is selected in addition to either a SYSTEM or RAM clock, then only a single clock is given.

Table 12-5. EPS Engine Description

```
  alias eps_start              = rch9<52>                    // set to start
  alias event_code[1:4]    = { rch9<0:6>, rch9<7:13>,rch9<14:20>, rch9<21:27> }        // micro-code
  alias cnt_gen              = rch9<28:31>                 // generic counter
  alias cnt_scan             = rch8<0:15>                  // scan counter
  alias slow_clk             = rcha<0:5>                   // sleep between clock
  alias scan_speed           = rch2<28:31>                 // sc1 speed
  alias loop_type            = rch9<55:57>                 // where to loop after last instr
  alias test_len             = rchb<0:19>                  // # of loops
  alias load_phase_imm  = rch9<53>                    // load phase prior to EPS run
  alias load_phase           = rch9<54>                    // load phase after each instr.
  alias mod48val             = rch9<36:37>rch9<32:35>  // phase value to load
  alias eps_running          = rch4<0>                     // running
  alias eps_complete         = rch4<1>                     // completed
  alias run_continuous       = rch2<22>                    // loops for ever
  alias eps_break             = rch9<58>                    // stop EPS machine

  waitfor (eps_start == 1);

  eps_running=1; eps_complete=0;
  pc=4;
  if(load_phase_imm) mod48 = mod48val;

loop1:
  event          = event_code[pc];
  c1_stop        = event[0];
  sc1_stop       = event[1];
  ramc2_stop     = event[2];
  c2star_stop    = event[3];
  killdynclk     = event[4];
  useslowclks    = event[5];

  if(       c1_stop == 1 AND sc1_stop == 0 AND event[6] == 1) cnt = cnt_gen;
  else if(c1_stop == 0 AND sc1_stop == 1                          ) cnt = cnt_scan;
  else cnt=0;

loop2:
  if(useslowclks) stop c1,sc1,ramc2,c2star clocks for slow_clk cycles;
  start clks for 1 cycle depending on c1_stop,sc1_stop,ramc2_stop,c2star_stop,killdynclk;
  if(sc1_stop) stop c1,sc1,ramc2,c2star clocks for scan_speed cycles;

  cnt--; if (cnt >= 0) goto loop2;

  pc--; if(pc==0)
  switch (loop_type)
     case '000': pc=4; // loop to 1st event
     case '100': pc=1; // loop to 4th event
     case '101': pc=3; // loop to 2nd event
     case '110': pc=2; // loop to 3rd event
  test_len--;

  if(load_phase) mod48 = mod48val;
  if((test_len>=0 OR continuous_run) AND NOT eps_break) goto loop1;
  eps_running=0; eps_complete=1;
```

Figure 12-7. Example of LBIST Commands using the EPS Engine

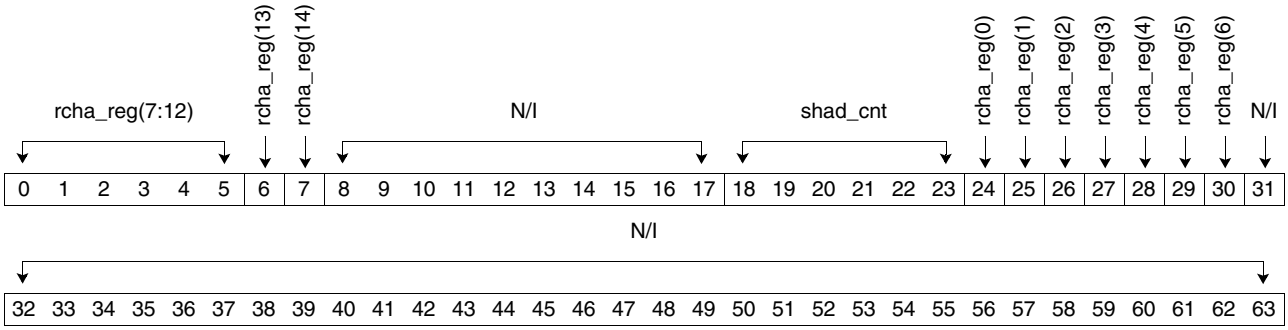
```
-- Scan / Wait / Clock 1 (All C1 Clocks) --
x‘840002’ x‘0000 0005 0000 0000’ Scan Clock Speed = 5
x‘600400’ x‘7000 0000 0000 0000’ N:1 Bus Ratios set to 1:1
x‘80000A’ x‘7800 000E 0000 0000’ slow clock=15
x‘84000B’ x‘vvvv v000 0000 0000’ Test Length Value: 2**20 max loops
x‘800009’ x‘B819 2040 0392 0802’ LBIST Clock Event


-- Scan / Wait / Clock 2 (First C1 clock is only 1:1) --
x‘840002’ x‘0000 0005 0000 0000’ Scan Clock Speed = 5
x‘600400’ x‘0000 0000 0000 0000’ N:1 Bus Ratios set to 2:1
x‘80000A’ x‘7800 000E 0000 0000’ slow clock=15
x‘84000B’ x‘vvvv v000 0000 0000’ Test Length Value: 2**20 max loops
x‘800009’ x‘BA19 2041 0392 0A03’ LBIST Clock Event (generic counter=1)


-- Scan / Wait / Clock 2 (N:1 Clocks occur with first 1:1 C1) --
x‘840002’ x‘0000 0005 0000 0000’ Scan Clock Speed = 5
x‘600400’ x‘0000 0000 0000 0000’ N:1 Bus Ratios set to 2:1
x‘80000A’ x‘7800 000E 0000 0000’ slow clock=15
x‘84000B’ x‘vvvv v000 0000 0000’ Test Length Value: 2**20 max loops
x‘800009’ x‘BA19 2041 1392 0A03’ LBIST Clock Event (generic counter=1)
```

Energy Star Register

Address x'80000A'

Type R/W

Reset Set to x'xx00 0x20 0000 0000' during POR (values marked as x depend on POR sequence).



Bits	Field Name	Description
0:5	rcha_reg(7:12)	Clock rate divide counter (0 to 5). Programmed count value: down counter. **Note:** Moved here from the Clock Command Control Register due to space limitation.
6	rcha_reg(13)	Enable debug logic on. The initial value is '0'.
7	rcha_reg(14)	Fuse clock control. The initial value is '0'.
8:17	N/I	Not implemented.
18:23	shad_cnt	Shadow counter *(Read-Only).* The master counter values at the time of the clock freeze on an immediate stop are in the shadow counter. This shadow counter value can be read and then loaded into the master mod48 counter to restart the clocks using Run-N in the event processor exactly where they were stopped.
24	rcha_reg(0)	Enter Energy Star mode.
25	rcha_reg(1)	Exit Energy Star mode.
26	rcha_reg(2)	Configuration shadow stop enable. Allow starting and pulsing clocks based on the saved value for the master phase hold counter. **Note:** In order for this mode to work, this bit must be written while both the STS and I/O clocks are running. From the POR state, the following sequence must be programmed: 1. x'80_00_00' x'0008_1800_0000_0000' Start *both* STS *and* I/O clocks. 2. x'80_00_0A' x'0000_0020_0000_0000' Enable shadow stop. 3. x'80_00_00' x'000C_1800_0000_0000' Stop *either* STS *or* I/O clocks. 4. x'80_00_09' x'0000_0000_0F38_1002' Scan0. 5. x'80_00_09' x'0000_0000_0000_0000' Clear array inhibit and receiver inhibit. 6. x'80_00_00' x'0004_D800_0000_0000' Pulse clocks (repeat as needed). ... 7. x'80_00_00' x'0008_D800_0000_0000' Run clocks. Clocks resume at the next, subsequent phase_hold alignment after the last pulse clock.
27	rcha_reg(3)	*Unconditional immediate exit* from Energy Star mode.
28	rcha_reg(4)	If '0', stops ABIST clocks in core and VMX.
29	rcha_reg(5)	If '0', stops ABIST clocks in STS.

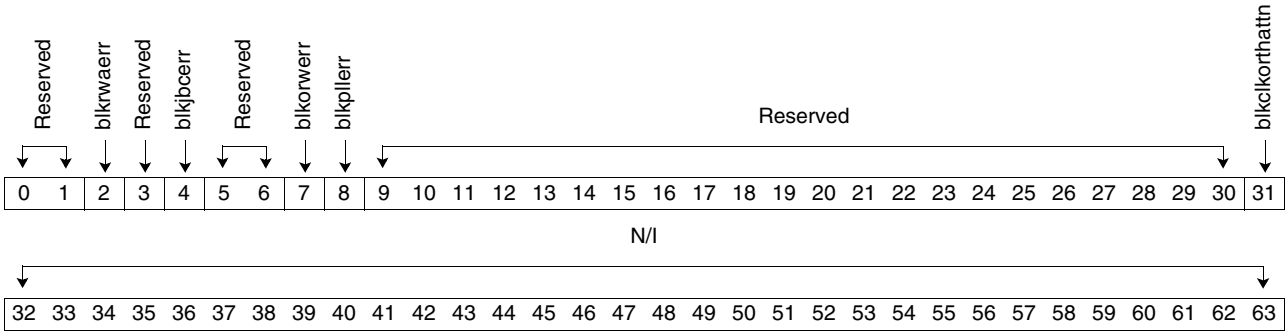
Bits	Field Name	Description
30	rcha_reg(6)	If '0', stops ABIST clocks in scanned ABIST machines. In addition, if '0' this bit prevents scanning through the local clock (lclk) domain.
31	N/I	Not implemented.
32:63	N/I	Not implemented.

Status Register Mask

Address x'80000C'

Type R/W

Reset Set to x'0080 0000 0000 0000' during POR.



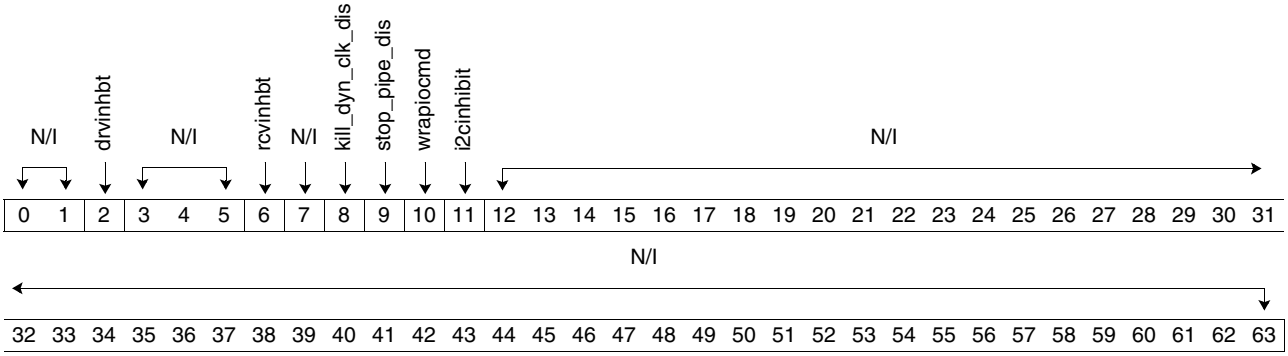
Bits	Field Name	Description
0:1	Reserved	Reserved.
2	blkrwaerr	Block invalid read/write address error.
3	Reserved	Reserved.
4	blkjbcerr	Block JTAG/BIST collision error.
5:6	Reserved	Reserved.
7	blkorwerr	Block Options Register write error.
8	blkpllerr	Block PLL lock error.
9:30	Reserved	Reserved.
31	blkclkorthattn	Block clock orthogonality check from generating attention.
32:63	N/I	Not implemented.

I/O Control Register

Address x'80000F'

Type R/W (Writing this register while I/O is running will raise an attention.)

Reset Set to x'2200 0000 0000 0000' during POR.



Bits	Field Name	Description
Note: The tristate control bits(0:6) determine whether drivers are placed in a high-impedance state. A bit value of '1' places the appropriate drivers in the high-impedance state. These bits should be set before executing LBIST, and are reset to all ones during POR.		
0:1	N/I	Not implemented.
2	drvinhbt	Drivers inhibit.
3:5	N/I	Not implemented.
6	rcvinhbt	Receivers inhibit.
7	N/I	Not implemented.
8	kill_dyn_clk_dis	This bit is asserted to stop the kill_dyn_clk (that is, it asserts the *kill_dyn_clk* signal). **Note:** This bit *must* be zero at all times.
9	stop_pipe_dis	This bit disables the global *stop_pipe* signal, which is used to enable the stop_ctl to dynamic logic. The *stop_pipe* signal is asynchronous. **Note:** This control turns on the clocks to the dynamic, nonscan L1 latches in order to initialize them after SCAN 0 initialization. This bit must be zero at all other times.
10	wrapiocmd	Prevents driver inhibit during the wrap I/O test.
11	i2cinhibit	I^2C inhibit. If set, the I^2C outputs to the JTAG macro are fenced. This bit is set when the LBIST bit, bit 46 in the Clock Command Control Register, is set (not reset when resetting LBIST bit).
12:63	N/I	Not implemented.

ABIST Status Register

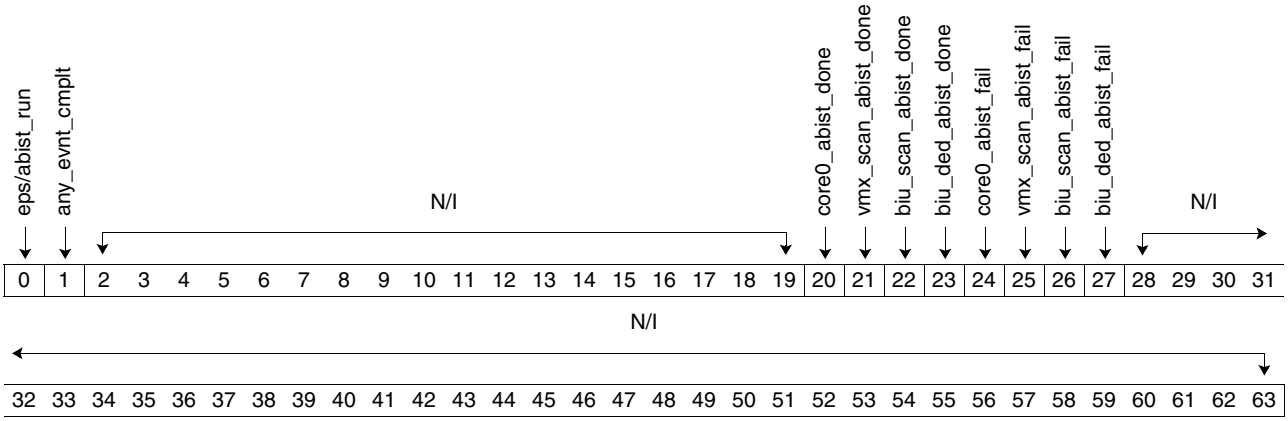
Address x'820004'

Type R/W

Bits 21:26 are unspecified after an LBIST SCAN0.SCAN and should be reinitialized before an ABIST. ABIST fail bits are only valid if the corresponding ABIST done bit is asserted.

Bits in this register are only updated during an EPS controlled operation. Turn on ccintf_local_psav_dis if using ABIST with functional clocks.

Reset Set to x'0000 0000 0000 0000' during POR.



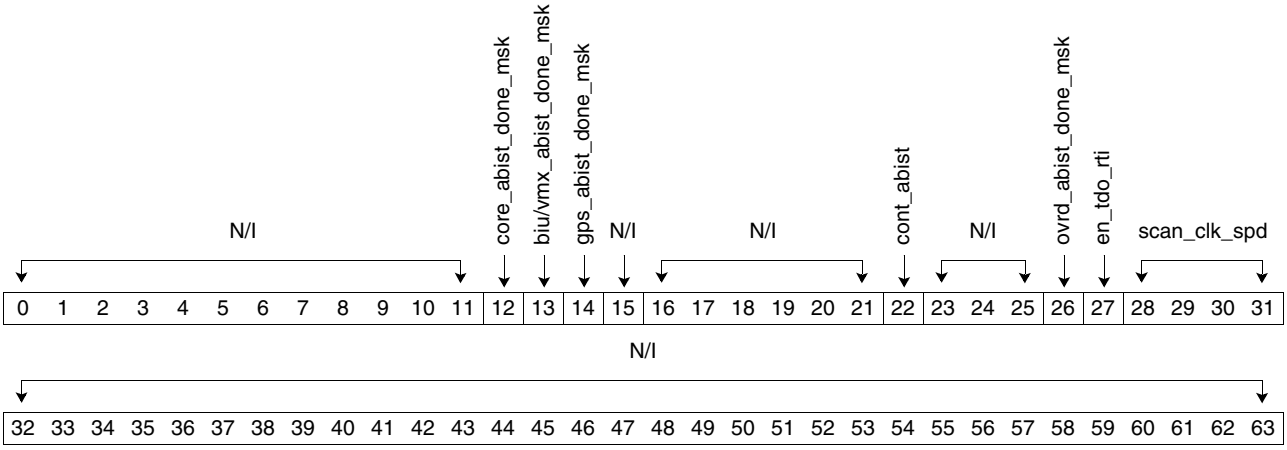
Bits	Field Name	Description
0	eps/abist_run	EPS running, or a decentral ABIST in progress. This is a read-only bit.
1	any_evnt_cmplt	Any event complete. Cleared by running an event. Set and held after completion of event processing. Cleared when bit 52 of the Clock Command Control Register is zero.
2:19	N/I	Not implemented.
20	core0_abist_done	Physical core0 (without VMX) ABIST done.
21	vmx_scan_ abist_done	VMX scanned; ABIST done.
22	biu_scan_ abist_done	BIU scanned; ABIST done.
23	biu_ded _abist_done	BIU dedicated; ABIST done.
24	core0_abist_fail	Core ABIST Fail. The ABIST Fail is held after being set. To clear this bit with an SCOM write to this register, the abist_out must be cleared first.
25	vmx_scan _abist_fail	VMX scanned; ABIST fail. The ABIST Fail is held after being set. To clear this bit with an SCOM write to this register, abist_out must be cleared first.
26	biu_scan _abist_fail	BIU scanned; ABIST fail. The ABIST Fail is held after being set. To clear this bit with an SCOM write to this register, abist_out must be cleared first.
27	biu_ded _abist_fail	BIU dedicated; ABIST fail. The ABIST Fail is held after being set. To clear this bit with an SCOM write to this register, abist_out must be cleared first.
28:63	N/I	Not implemented.

LBIST Options Register

Address x'840002'

Type R/W

Reset Set to x'000E 0005 0000 0000' during POR.



Bits	Field Name	Description
0:11	N/I	Not implemented.
12	core_abist_done_msk	Physical core ABIST done mask. Mask ABIST done from the core. (ABIST pass is also masked as a result of the masking.)
13	biu/vmx_abist_done_msk	BIU + VMX scanned ABIST done mask.
14	gps_abist_done_msk	STS ABIST done mask. Mask ABIST done from STS. (ABIST pass is also masked as a result of the masking.)
15:21	N/I	Not implemented.
22	cont_abist	Continual LBIST. Set in order to override the LBIST Test Length Register. Useful for power and noise measurements. Continual LBIST is stopped with a write to the eps_override bit (bit 58) of the *Clock Command Register* (x'800009').
23:25	N/I	Not implemented.
26	ovrd_abist_done_msk	Override ABIST done mask. The ABIST pass bit is gated by not *abist_done*. If *abist_done* is not asserted, then the ABIST pass status cannot be determined. However, by setting this override bit, you can observe the ABIST pass status. **Notes:** • The *abist_fail* is '0' until *abist_done* is set. Once *abist_done* is set, then *abist_fail* may be set. • This override effects all three domains: core, scanned ABIST not core, and STS. **Programming Note:** The gps_scan_abist requires x'FFFFF' + 1 number of tester loops to complete. Each tester loop is 131 times the Scan_Clock_Rate number of mesh clocks. Thus, only x'4FFFF' tester loops are run. Therefore, the *abist_done* status bit will not be set, and the Override ABIST DONE Mask should be set in order to validate the ABIST_FAIL status. The *abist_done* signal from the GPS_SCAN_ABIST engine (HTBC RLM) should be checked.

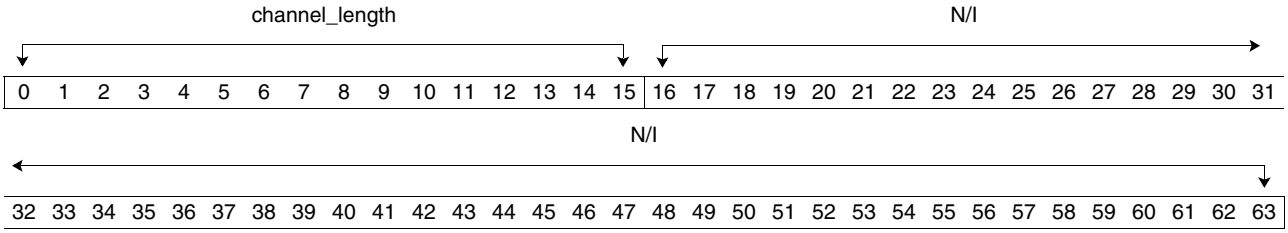
Bits	Field Name	Description
27	en_tdo_rti	Enable TDO during *RUN_TEST_IDLE* (for ABIST real time fail observation). The bit-fail mapping and array diagnostics for debugging ABIST fails on the tester often need to observe the ABIST fail on a real-time basis. Because the ABIST fail information is sent to ACCESS for status, it is conveniently routed to TDO, which is typically a burn-in pin as well. A special *TDO_ENABLE* is required to observe this signal. **Note:** The normal *IEEE JTAG Specification* has the TDO in a high-impedance state during run-test-idle. This state is then used to enable the *TDO_ENABLE* during ABIST operations when bit 27 is asserted.
28:31	scan_clk_spd	Scan clock speed. Set to the divide-by value to limit scan clock frequency during scan events on the event processor. This accommodates relaxed timing on the scan paths. Typical applications are flush0, LBIST, and scanned ABIST. The scan clock speed has no effect on system scan-ring access using the 0F access command, which occurs with TCK frequency when in the SHIFT_DR state. Bit 31 is the LSb. • The nominal value is '0101' (that is, 1/6th of the system clock). • A value of '0000' *is* supported, and can be used for increased simulation throughput. • The value x'0001' is invalid and must not be used.
32:63	N/I	Not implemented.

LBIST Channel Length Register

Address x'840008'

Type R/W

Reset Set to x'0800 0000 0000 0000' during POR.



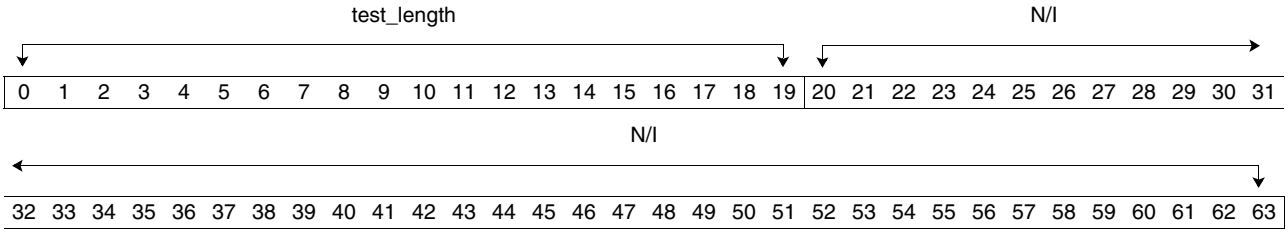
Bits	Field Name	Description
0:15	channel_length	Channel length. Contains the number of cycles *minus one* the event processor is to loop on a particular event that has the SCAN_CTL bit asserted in the microcoded instructions. **Exception:** When RAMSTOP_CTL is set coincident with SCAN_CTL, only one scan clock pulse and one ramc2 clock pulse are generated together after a delay of the SCAN_SPEED value from the beginning of the event.
16:63	N/I	Not implemented.

LBIST Test Length Register

Address x'84000B'

Type R/W

Reset Set to x'4000 0000 0000 0000' during POR.



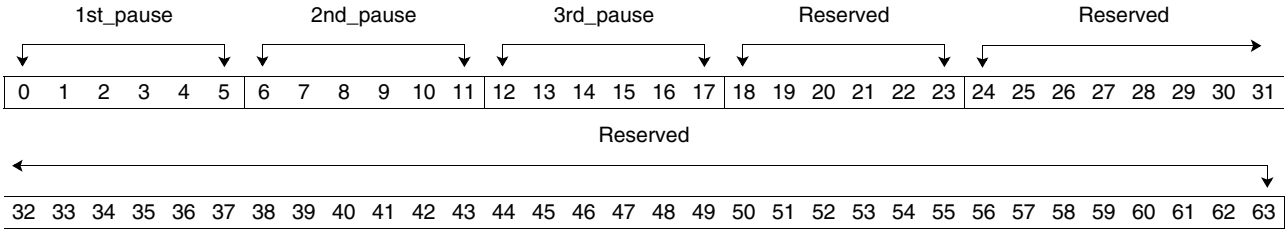
Bits	Field Name	Description
0:19	test_length	Test length. Contains the number of cycles *minus one* that the event processor is to loop on the micro-coded instructions. For LBIST, it would be the number of pseudo-random test pattern generator (PRPG) loads and system clock loops to run. If loaded to all zeros, then *one* PRPG load and system clock cycle is run. MISR CLEAR has no overriding effect on the test length. **Example:** To loop four times, program 3 into this register ('0000000000000000011'). **Note:** Do not attempt less than two loops in this register while doing a RUN_N that is using the last event due to logic implementation (pipelining).
20:63	N/I	Not implemented.

Clock Ramping Configuration Register

Address x'84000D'

Type R/W

Reset Set to x'2D75 E300 0000 0000' during POR.



Bits	Field Name	Description
0:5	1st_pause	First pause in ramping up or down. 5 cycles at f, 2 cycles at f/2, 1 cycle at f/4
6:11	2nd_pause	Second pause in ramping up or down. 21 cycles at f, 10 cycles at f/2, 5 cycles at f/4
12:17	3rd_pause	Third pause in ramping up or down. 26 cycles at f, 13 cycles at f/2, 6 cycles at f/4
18:23	Reserved	Reserved.
24:63	N/I	Not implemented.

13. Vector Processing Unit

The Vector/SIMD technology, referred to in this document as the vector processing unit (VPU), provides a software model that accelerates the performance of various software applications and runs on reduced instruction set computing (RISC) microprocessors. This is a short vector parallel architecture that extends the instruction set architecture (ISA) of the PowerPC Architecture. It is based on separate vector/SIMD[1]-style execution units that have high data parallelism. This parallelism allows it to perform on multiple data elements in a single instruction. The term vector refers to the spatial parallel processing of short, fixed-length, one-dimensional matrices performed by an execution unit. It should not be confused with the temporal parallel (pipelined) processing of long, variable-length vectors performed by classical vector machines. High degrees of parallelism are achievable with simple in-order instruction dispatch and low-instruction bandwidth. However, the ISA is designed so as not to impede additional parallelism through superscalar dispatch to multiple execution units or multithreaded execution unit pipelines.

13.1 970MP Vector and SIMD Multimedia Overview

The VPU expands the current PowerPC Architecture through the addition of a 128-bit vector execution unit, which operates concurrently with the existing scalar integer and floating-point units. This new engine provides for highly parallel operations, allowing for the simultaneous execution of up to four 32-bit floating operations or sixteen 8-bit fixed-point operations in one instruction. All VPU data paths and execution units are 128 bits wide and are fully pipelined.

13.1.1 VPU Implementation

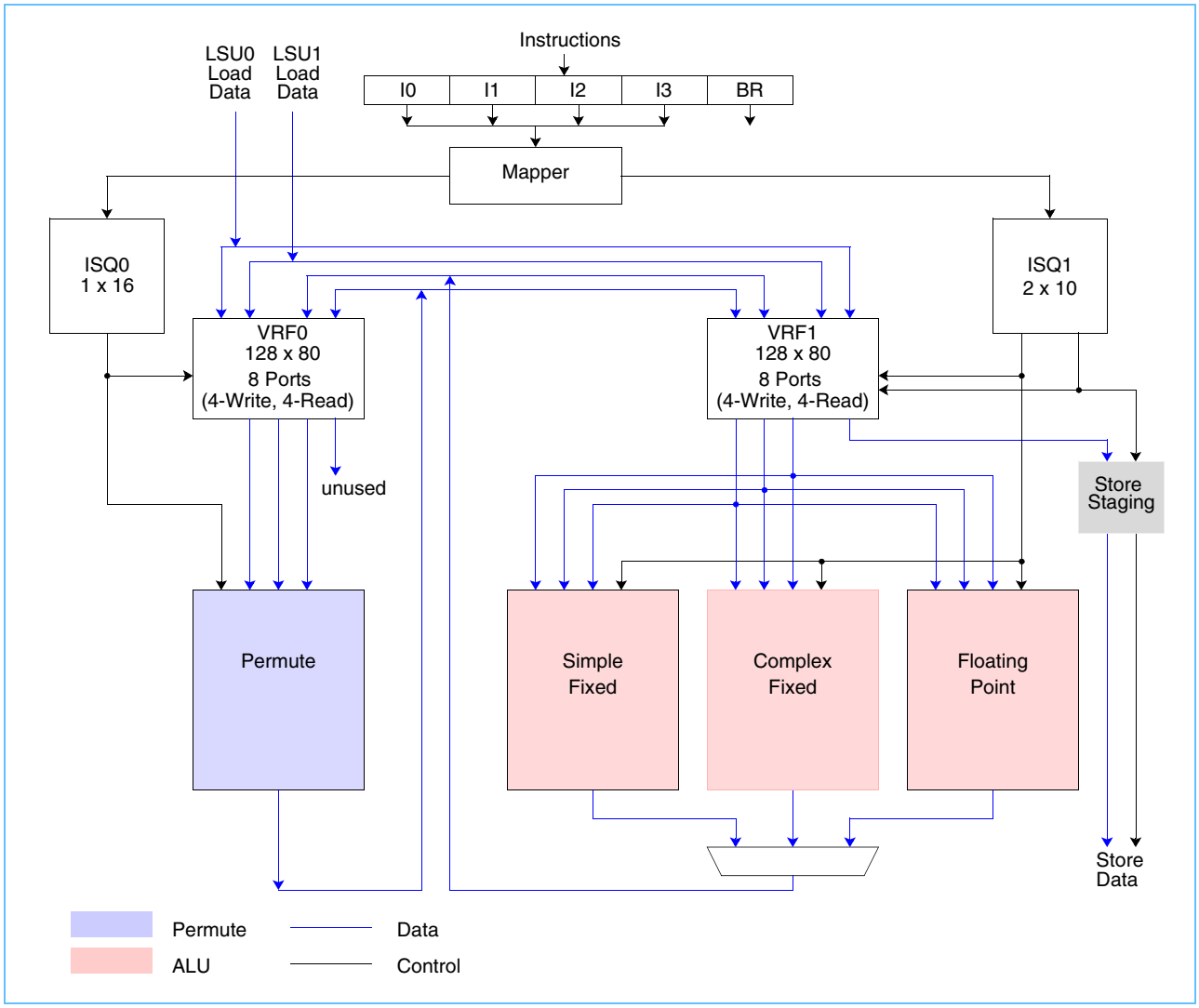
The 970MP microprocessor implements many aspects of a preferred implementation as described in the *PowerPC Microprocessor Family: Vector/SIMD Multimedia Extension Technology Programming Environments Manual*. The key features of a preferred implementation include:

- All data paths and execution units are 128 bits wide.
- There are two independent VPU sub-units, one for all arithmetic logic unit (ALU) instructions and one for permute operations.

The VPU is divided into two dispatchable units: vector ALU and vector permute. The vector ALU unit is further subdivided into a vector floating-point unit, a vector simple-fixed unit, and a vector complex-fixed unit. The vector ALU and permute units receive predecoded instructions from the issue queue in the instruction sequencer unit for the VPU (ISV). Vector instructions are issued to the appropriate vector unit when all of the source operands are available. Vector loads, stores, and data stream touch (DST) instructions are executed in the load/store unit (LSU) pipes. There are two copies of the Vector Register files; one provides operands for the vector permute unit, and one provides operands for the vector ALU. *Figure 13-1* on page 492 provides a high-level view of the instruction sequencer unit (ISU) interaction with the VPUs.

1. Single instruction stream, multiple data streams

Figure 13-1. VPU Block Diagram



13.1.2 Vector ALU

Conceptually, the vector unit ALU is capable of operating on three source vectors and producing a single result vector on each instruction. The ALU is an SIMD-style arithmetic unit, where an instruction performs the same operation on all the data elements that comprise each vector. The ALU is partitioned into four separate ALUs for 32-bit integers and for single-precision floating-point operands. For 16-bit integers, the ALU is partitioned into 8 ALUs, and for 8-bit integers it is partitioned into 16 separate ALUs. No arithmetic is performed on elements larger than 32 bits. The largest adder in the vector ALU is 32 bits wide, and the largest multiplier array is 24 bits wide for the single-precision floating-point mantissa.

13.2 Vector Registers

13.2.1 VRSAVE Register

This 32-bit register is maintained and managed by software only. Each bit in the VRSAVE Register corresponds to a Vector Register and indicates whether the corresponding register is currently being used by the executing process. Therefore, the operating system needs to save and restore only those Vector Registers (VRs) when an exception occurs. The register is handled as a renamed register within the General Purpose Register (GPR) file in the 970MP microprocessor (see *Section 6.3.5 Register Renaming* on page 183 for more information).

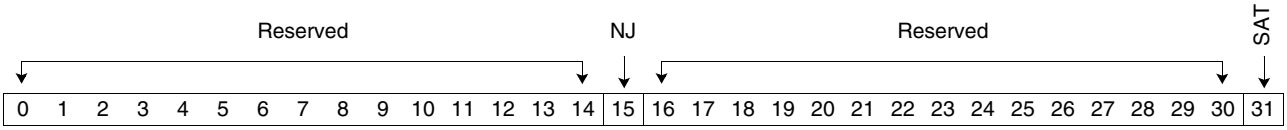
Note: If this approach is taken, it must be applied rigorously. If a program fails to indicate that a given Vector Register is in use, software errors may occur that will be difficult to detect and correct because they are timing-dependent. Some operating systems save and restore VRSAVE only for programs that also use other vector registers.

The VRSAVE Register can be accessed only by the Move From Special Purpose Register (**mfspr**) or Move To Special Purpose Register (**mtspr**) instructions. The **mfspr** instruction copies VRSAVE to the low-order 32 bits of a GPR; the **mtspr** instruction copies the low-order 32 bits of a GPR to VRSAVE.

13.2.2 Vector Status and Control Register (VSCR)

The Vector Status and Control Register is a special 32-bit register (not an SPR) that is read and written in a manner similar to the Floating-Point Status and Control Register (FPSCR) in the scalar floating-point unit. Two special instructions, Move From Vector Status and Control Register (**mfvscr**) and Move To Vector Status and Control Register (**mtvscr**), are provided to move the VSCR from and to a Vector Register. When moved to or from a Vector Register, the 32-bit VSCR is right justified in the 128-bit Vector Register. When moved to a Vector Register, the upper 96 bits (0:95) of the Vector Register are cleared (set to zeros).

Figure 13-2. VSCR Format



The VSCR has two defined bits, the non-Java mode (NJ) bit (VSCR[15]) and the saturation (SAT) bit (VSCR[31]). The remaining bits are reserved. VSCR bit settings are shown in *Table 13-1* on page 494.

The VSCR bits, after being moved to a Vector Register, are shown in *Figure 13-3*.

Figure 13-3. VSCR Moved to a Vector Register

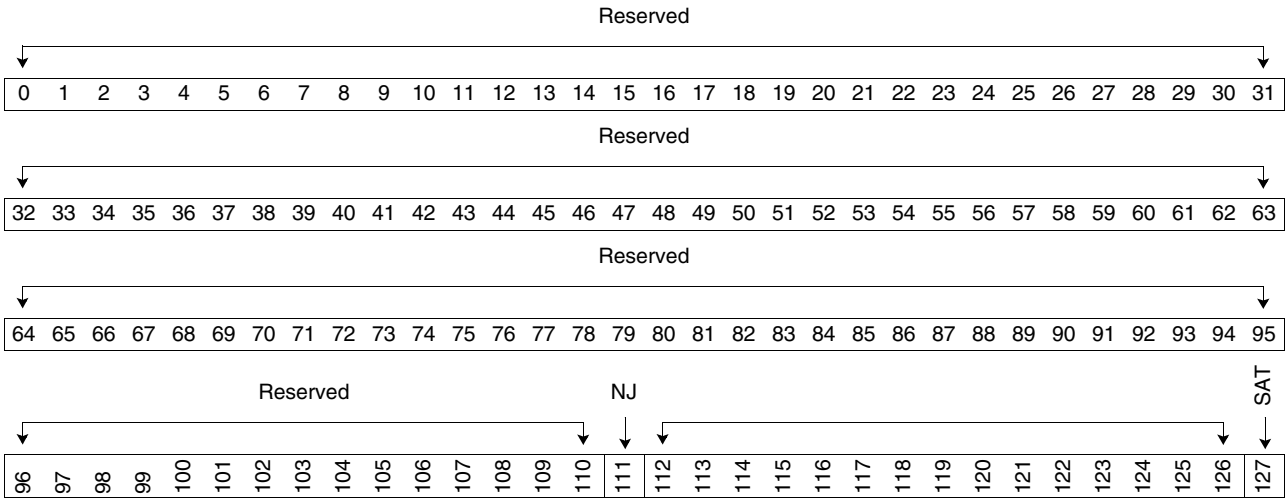


Table 13-1. VSCR Field Descriptions

Bits	Field Name	Description
0:14	—	Reserved.
15	NJ	Non-Java. A mode control bit that determines whether VPU floating-point operations are performed in a mode that is compliant with Java, IEEE, and C9X or in a possibly faster noncompliant mode. 0 The Java-IEEE-C9X–compliant mode is selected. Denormalized values are handled as specified by the Java, IEEE, and C9X standard. 1 The non-Java/non-IEEE–compliant mode is selected. If an element in a source Vector Register contains a denormalized value, the value '0' is used instead. If an instruction causes an underflow exception, the corresponding element in the target Vector Register (VR) is cleared to '0'. In both cases, the '0' has the same sign as the denormalized or underflowing value.
16:30	—	Reserved.
31	SAT	Saturation. A sticky status bit indicating that some field in a saturating instruction became saturated since the last time SAT was cleared. In other words, when SAT is set to '1', it remains set to '1' until it is cleared to '0' by a **mtvscr** instruction. 0 Indicates no saturation has occurred since this bit was last cleared by a **mtvscr** instruction. 1 The vector saturate instruction implicitly sets when saturation has occurred on the results of one of the vector instructions having saturate in its name. (See *Table 13-4* on page 499.)

The **mtvscr** instruction is context synchronizing. This implies that all vector instructions logically preceding an **mtvscr** in the program flow will execute in the architectural context (NJ mode) that existed before completion of the **mtvscr**. It also implies that all instructions logically following the **mtvscr** will execute in the new context (NJ mode) established by the **mtvscr**.

After an **mfvscr** instruction executes, the result in the target Vector Register is architecturally precise. It reflects all updates to the SAT bit that could have been made by vector instructions logically preceding it in the program flow. Further, it will not reflect any SAT updates that may be made to it by vector instructions logically following it in the program flow. Reading the VSCR can be much slower than typical vector instructions, and therefore care must be taken in reading it to avoid performance problems.

13.3 Effects on Existing PowerPC Facilities

13.3.1 Control Flow

Vector instructions can be freely intermixed with existing PowerPC instructions to form a complete program. Vector instructions provide a vector compare and select mechanism to implement conditional execution as the preferred mechanism to control data flow in VPU programs. Vector compare instructions can update the Condition Register, thus providing the communication from the vector execution units to the PowerPC branch instructions necessary to modify program flow based on vector data.

13.3.1.1 Condition Register

The Condition Register (CR) is affected by the VPU architecture. The CR is a 32-bit register, divided into eight 4-bit fields, CR0-CR7 (see *Figure 13-4*), that reflect the results of certain arithmetic operations and provide a mechanism for testing and branching. For the VPU ISA, the CR6 field can optionally be used. If the record bit (Rc) of a vector instruction field is set in a vector compare instruction, then the CR6 field is updated according to *Table 13-2*.

Figure 13-4. Condition Register (CR)

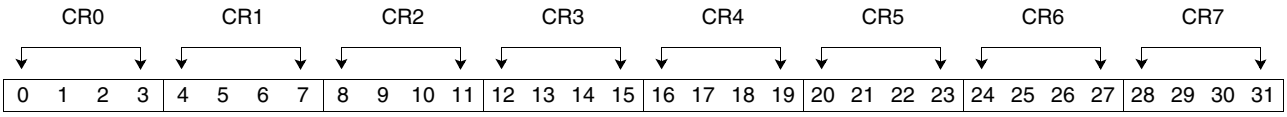


Table 13-2. CR6 Field Bit Settings for Vector Compare Instructions

Bits	Description
	Vector Compare
0	1 Comparison successful for all fields 0 Comparison failed for at least one field
1	Always zero
2	1 Comparison failed for all fields 0 Comparison successful for at least one field
3	Always zero
	Vector Compare Bounds (vcmpbfp)
0	Always zero
1	Always zero
2	1 All values within bounds 0 Not all values within bounds
3	Always zero

The Rc bit should be used sparingly. As for other PowerPC instructions, in some implementations, instructions with the Rc bit set to '1' could have a longer latency or be more disruptive to the instruction pipeline flow than instructions with the Rc bit set to '0'. Therefore, techniques of accumulating results and testing infrequently are advised.

13.3.1.2 Machine State Register

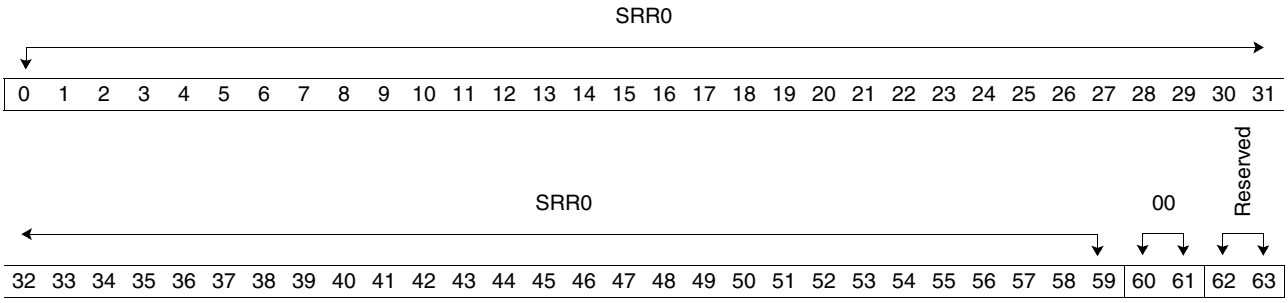
Certain bits in the Machine State Register (MSR) affect instructions in the vector data stream. MSR[VP] indicates whether the vector processor is available. *Table 13-3* defines the VP, PR, and DR bits.

Table 13-3. MSR Bit Settings Affecting the VPU

Bits	Field Name	Description
38	VP	VP available. 0 The processor prevents execution of all vector instructions, including loads, stores, and moves. If such execution is attempted, a VPU unavailable exception is raised. 1 The processor can execute all vector instructions. **Note:** The VRSAVE Register is not protected by MSR[VP]. The data streaming family of instructions (**dst**, **dstt**, **dstst**, **dststt**, **dss**, and **dssall**) are not affected by the MSR[VP].
49	PR	Problem (user) state. 0 The processor is privileged to execute any instruction. 1 The processor can only execute non-privileged instructions. **Note:** Care should be taken if data-stream prefetching is used in a supervisor (privileged) state (MSR[PR] is set to '0'). For each existing data stream, prefetching is enabled if (a) MSR[DR] is set to '1' and (b) MSR[PR] has the value it had when the **dst** or **dstst** instruction that specified the data stream was executed. Otherwise, prefetching for the data stream is suspended.
59	DR	Data address translation. 0 Data address translation is disabled. If data stream touch (**dst**) and data stream touch for store (**dstst**) instructions are executed when DR is set to '0', the results are boundedly undefined. 1 Data address translation is enabled. Data stream touch (**dst**) and data stream touch for store (**dstst**) instructions are supported when DR is set to '1'.

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

SRR0 holds the effective address (EA) for the instruction that caused the VPU unavailable exception, and SRR1 holds the machine state status as described in *Chapter 4 Exceptions*.



13.4 Exceptions

There are three exceptions that can result from the execution of a vector instruction:

- VPU unavailable exception
- VPU assist exception
- Data storage exception

13.4.1 VPU Unavailable Exception

This interrupt is described in *Section 13.3.1.2 Machine State Register* on page 496.

13.4.2 VPU Assist Exception

The VPU assist exception happens when operating in Java mode and either the input operands or the result of an operation are denormalized. After this exception, execution resumes at offset x'0000 0000 0000 1700'. See *Section 4.5.18 VPU Assist Exception* on page 145 for more information.

13.4.3 Data Storage Exception

Load Vector Indexed and Store Vector Indexed instructions transfer quadword vectors between memory and Vector Registers. Load Vector Element Indexed and Store Vector Element Indexed instructions transfer byte, halfword, and word scalar elements between memory and Vector Registers. All vector loads and vector stores use the index (**r**A|0 + **r**B) addressing mode to specify the target memory address. No update forms are provided. A Load Vector Element Indexed instruction transfers a scalar data element from memory into the destination Vector Register, leaving other elements in the vector with boundedly-undefined values. A Store Vector Element Indexed transfers a scalar data element from the source Vector Register to memory leaving other elements in the quadword unchanged. No data alignment occurs; that is, all scalar data elements are transferred directly on their natural memory byte-lanes to or from the corresponding element in the Vector Register. Quadword memory accesses made by Load Vector Indexed and Store Vector Indexed are not guaranteed to be atomic. Direct-store segments (where T equals '1') are not supported. Any vector load or store that attempts to access a direct-store segment will cause a data storage exception (DSI).

13.5 Optional Instructions

The 970MP microprocessor implements all vector instructions as listed in the *PowerPC Microprocessor Family: Vector/SIMD Multimedia Extension Technology Programming Environments Manual*. See *Table 13-4 Supported Vector Instructions* on page 499.

13.5.1 Java Mode Instruction Handling Implementation

The 970MP VPU implementation handles certain instructions differently based on the Java mode setting in the VSCR. Java compliance does require compliance with certain aspects of the IEEE Standard including:

- Support of denorms as inputs and results (gradual underflow) for arithmetic operations
- Not a number (NaN) results for invalid operations
- NaNs compare unordered with respect to everything, so that the result of any comparison of any NaN to any data type is always false

- NaNs are handled the same way in both the Java or non-Java mode for the 970MP implementation.

For some instructions, denormalization produces the exact result without trapping. The 970MP implementation of the VPU handles most denormalization by trapping at interrupt vector x'0000 0000 0000 1700' (VPU assist interrupt).

13.5.2 Least Recently Used Instructions

The Vector/SIMD Architecture suggests that Load Vector Indexed LRU (**lvxl**) and Store Vector Indexed LRU (**stvxl**) are handled differently than the regular load/store instructions in that they leave cache entries in the least recently used (LRU) state instead of a most recently used (MRU) state. This supports efficient processing of data that is known to have little reuse and poor caching characteristics.

The 970MP microprocessor will treat **lvxl** and **stvxl** as a regular load and store with respect to the replacement algorithm. That is, the cache entry will be set as MRU.

13.5.3 Data Stream Instructions

DST instructions are broken up into two internal instructions (IOPs), one for the effective address and one for the prefetch information. They are marked serialized and are not executed until they are the next instruction to complete. Additional information can be found in *Section 3.5.4 Data Prefetch* on page 110.

13.6 Vector Instruction Set

Table 13-4 lists the supported vector instructions.

Table 13-4. Supported Vector Instructions (Page 1 of 7)

Number	Mnemonic	Operands	Ex-Unit	Description
Load Vector Element Indexed				
1	lvebx	**v**D,**r**A,**r**B	LOAD	Load Vector Element Byte Indexed
2	lvehx	**v**D,**r**A,**r**B	LOAD	Load Vector Element Halfword Indexed
3	lvewx	**v**D,**r**A,**r**B	LOAD	Load Vector Element Word Indexed
4	lvx	**v**D,**r**A,**r**B	LOAD	Load Vector Indexed
5	lvxl	**v**D,**r**A,**r**B	LOAD	Load Vector Indexed LRU
Store Vector Element Indexed				
6	stvebx	**v**S,**r**A,**r**B	STORE	Store Vector Element Byte Indexed
7	stvehx	**v**S,**r**A,**r**B	STORE	Store Vector Element Halfword Indexed
8	stvewx	**v**S,**r**A,**r**B	STORE	Store Vector Element Word Indexed
9	stvx	**v**S,**r**A,**r**B	STORE	Store Vector Indexed
10	stvxl	**v**S,**r**A,**r**B	STORE	Store Vector Indexed LRU
Load Vector for Shift				
11	lvsl	**v**D,**r**A,**r**B	LOAD	Load Vector for Shift Left
12	lvsr	**v**D,**r**A,**r**B	LOAD	Load Vector for Shift Right

Table 13-4. Supported Vector Instructions (Page 2 of 7)

Number	Mnemonic	Operands	Ex-Unit	Description
Move To and Move From Vector Status and Control Register				
13	mtvscr	**v**B	Simple	Move To Vector Status and Control Register
14	mfvscr	**v**D	Simple	Move From Vector Status and Control Register
Data Stream				
15	dst	**r**A,**r**B,tag	LSU	Data Stream Touch
16	dstt	**r**A,**r**B,tag	LSU	Data Stream Touch Transient (treated as dst)
17	dstst	**r**A,**r**B,tag	LSU	Data Stream Touch for Store (treated as dst)
18	dststt	**r**A,**r**B,tag	LSU	Data Stream Touch for Store Transient (treated as dst)
19	dss	tag	LSU	Data Stream Stop
20	dssall		LSU	Data Stream Stop All
Vector Add				
21	vaddubm	**v**D,**v**A,**v**B	Simple	Vector Add Unsigned Byte Modulo
22	vaddubs	**v**D,**v**A,**v**B	Simple	Vector Add Unsigned Byte Saturate
23	vaddsbs	**v**D,**v**A,**v**B	Simple	Vector Add Signed Byte Saturate
24	vadduhm	**v**D,**v**A,**v**B	Simple	Vector Add Unsigned Halfword Modulo
25	vadduhs	**v**D,**v**A,**v**B	Simple	Vector Add Unsigned Halfword Saturate
26	vaddshs	**v**D,**v**A,**v**B	Simple	Vector Add Signed Halfword Saturate
27	vadduwm	**v**D,**v**A,**v**B	Simple	Vector Add Unsigned Word Modulo
28	vadduws	**v**D,**v**A,**v**B	Simple	Vector Add Unsigned Word Saturate
29	vaddsws	**v**D,**v**A,**v**B	Simple	Vector Add Signed Word Saturate
30	vaddfp	**v**D,**v**A,**v**B	Float	Vector Add Float
Vector Add & Write Carry-Out				
31	vaddcuw	**v**D,**v**A,**v**B	Simple	Vector Add and Write Carry-Out Unsigned Word
Vector Subtract				
32	vsububm	**v**D,**v**A,**v**B	Simple	Vector Subtract Unsigned Byte Modulo
33	vsububs	**v**D,**v**A,**v**B	Simple	Vector Subtract Unsigned Byte Saturate
34	vsubsbs	**v**D,**v**A,**v**B	Simple	Vector Subtract Signed Byte Saturate
35	vsubuhm	**v**D,**v**A,**v**B	Simple	Vector Subtract Unsigned Halfword Modulo
36	vsubuhs	**v**D,**v**A,**v**B	Simple	Vector Subtract Unsigned Halfword Saturate
37	vsubshs	**v**D,**v**A,**v**B	Simple	Vector Subtract Signed Halfword Saturate
38	vsubuwm	**v**D,**v**A,**v**B	Simple	Vector Subtract Unsigned Word Modulo
39	vsubuws	**v**D,**v**A,**v**B	Simple	Vector Subtract Unsigned Word Saturate
40	vsubsws	**v**D,**v**A,**v**B	Simple	Vector Subtract Signed Word Saturate
41	vsubfp	**v**D,**v**A,**v**B	Float	Vector Subtract Float
Vector Subtract and Write Carry-Out				
42	vsubcuw	**v**D,**v**A,**v**B	Simple	Vector Subtract and Write Carry-Out Unsigned Word

Table 13-4. Supported Vector Instructions (Page 3 of 7)

Number	Mnemonic	Operands	Ex-Unit	Description
Vector Multiply Odd Integer				
43	vmuloub	**v**D,**v**A,**v**B	Complex	Vector Multiply Odd Unsigned Byte
44	vmulosb	**v**D,**v**A,**v**B	Complex	Vector Multiply Odd Signed Byte
45	vmulouh	**v**D,**v**A,**v**B	Complex	Vector Multiply Odd Unsigned Halfword
46	vmulosh	**v**D,**v**A,**v**B	Complex	Vector Multiply Odd Signed Halfword
Vector Multiply Even Integer				
47	vmuleub	**v**D,**v**A,**v**B	Complex	Vector Multiply Even Unsigned Byte
48	vmulesb	**v**D,**v**A,**v**B	Complex	Vector Multiply Even Signed Byte
49	vmuleuh	**v**D,**v**A,**v**B	Complex	Vector Multiply Even Unsigned Halfword
50	vmulesh	**v**D,**v**A,**v**B	Complex	Vector Multiply Even Signed Halfword
Vector Multiply-Add				
51	vmhaddshs	**v**D,**v**A,**v**B,**v**C	Complex	Vector Multiply-High and Add Signed Halfword Saturate
52	vmhraddshs	**v**D,**v**A,**v**B,**v**C	Complex	Vector Multiply-High Round and Add Signed Halfword Saturate
53	vmladduhm	**v**D,**v**A,**v**B,**v**C	Complex	Vector Multiply-Low and Add Unsigned Halfword Modulo
54	vmaddfp	**v**D,**v**A,**v**C,**v**B	Float	Vector Multiply-Add Float
Vector Multiply-Sum Integer				
55	vmsumubm	**v**D,**v**A,**v**B,**v**C	Complex	Vector Multiply-Sum Unsigned Byte Modulo
56	vmsummbm	**v**D,**v**A,**v**B,**v**C	Complex	Vector Multiply-Sum Mixed-Sign Byte Modulo
57	vmsumuhm	**v**D,**v**A,**v**B,**v**C	Complex	Vector Multiply-Sum Unsigned Halfword Modulo
58	vmsumuhs	**v**D,**v**A,**v**B,**v**C	Complex	Vector Multiply-Sum Unsigned Halfword Saturate
59	vmsumshm	**v**D,**v**A,**v**B,**v**C	Complex	Vector Multiply-Sum Signed Halfword Modulo
60	vmsumshs	**v**D,**v**A,**v**B,**v**C	Complex	Vector Multiply-Sum Signed Halfword Saturate
Vector Sum Across Signed Integer Saturate				
61	vsumsws	**v**D,**v**A,**v**B	Complex	Vector Sum Across Signed Word Saturate
Vector Sum Across Partial (1/2) Signed Integer Saturate				
62	vsum2sws	**v**D,**v**A,**v**B	Complex	Vector Sum Across Partial (1/2) Signed Word Saturate
Vector Sum Across Partial (1/4) Integer Saturate				
63	vsum4ubs	**v**D,**v**A,**v**B	Complex	Vector Sum Across Partial (1/4) Unsigned Byte Saturate
64	vsum4sbs	**v**D,**v**A,**v**B	Complex	Vector Sum Across Partial (1/4) Signed Byte Saturate
65	vsum4shs	**v**D,**v**A,**v**B	Complex	Vector Sum Across Partial (1/4) Signed Halfword Saturate
Vector Average Integer				
66	vavgub	**v**D,**v**A,**v**B	Simple	Vector Average Unsigned Byte
67	vavgsb	**v**D,**v**A,**v**B	Simple	Vector Average Signed Byte
68	vavguh	**v**D,**v**A,**v**B	Simple	Vector Average Unsigned Halfword
69	vavgsh	**v**D,**v**A,**v**B	Simple	Vector Average Signed Halfword
70	vavguw	**v**D,**v**A,**v**B	Simple	Vector Average Unsigned Word
71	vavgsw	**v**D,**v**A,**v**B	Simple	Vector Average Signed Word

Table 13-4. Supported Vector Instructions (Page 4 of 7)

Number	Mnemonic	Operands	Ex-Unit	Description
Vector Logical				
72	vand	**v**D,**v**A,**v**B	Simple	Vector Logical AND
73	vor	**v**D,**v**A,**v**B	Simple	Vector Logical OR
74	vxor	**v**D,**v**A,**v**B	Simple	Vector Logical XOR
75	vandc	**v**D,**v**A,**v**B	Simple	Vector Logical AND with Complement
76	vnor	**v**D,**v**A,**v**B	Simple	Vector Logical NOR
Vector Rotate Left Integer				
77	vrlb	**v**D,**v**A,**v**B	Simple	Vector Rotate Left Integer Byte
78	vrlh	**v**D,**v**A,**v**B	Simple	Vector Rotate Left Integer Halfword
79	vrlw	**v**D,**v**A,**v**B	Simple	Vector Rotate Left Integer Word
Vector Shift Left Integer				
80	vslb	**v**D,**v**A,**v**B	Simple	Vector Shift Left Integer Byte
81	vslh	**v**D,**v**A,**v**B	Simple	Vector Shift Left Integer Halfword
82	vslw	**v**D,**v**A,**v**B	Simple	Vector Shift Left Integer Word
83	vsl	**v**D,**v**A,**v**B	Simple	Vector Shift Left
Vector Shift Right Integer				
84	vsrb	**v**D,**v**A,**v**B	Simple	Vector Shift Right Byte
85	vsrab	**v**D,**v**A,**v**B	Simple	Vector Shift Right Algebraic Byte
86	vsrh	**v**D,**v**A,**v**B	Simple	Vector Shift Right Halfword
87	vsrah	**v**D,**v**A,**v**B	Simple	Vector Shift Right Algebraic Halfword
88	vsrw	**v**D,**v**A,**v**B	Simple	Vector Shift Right Word
89	vsraw	**v**D,**v**A,**v**B	Simple	Vector Shift Right Algebraic Word
90	vsr	**v**D,**v**A,**v**B	Simple	Vector Shift Right
Vector Compare Greater-Than				
91	vcmpgtub[.]	**v**D,**v**A,**v**B	Simple	Vector Compare Greater-Than Unsigned Byte [Record]
92	vcmpgtsb[.]	**v**D,**v**A,**v**B	Simple	Vector Compare Greater-Than Signed Byte [Record]
93	vcmpgtuh[.]	**v**D,**v**A,**v**B	Simple	Vector Compare Greater-Than Unsigned Halfword [Record]
94	vcmpgtsh[.]	**v**D,**v**A,**v**B	Simple	Vector Compare Greater-Than Signed Halfword [Record]
95	vcmpgtuw[.]	**v**D,**v**A,**v**B	Simple	Vector Compare Greater-Than Unsigned Word [Record]
96	vcmpgtsw[.]	**v**D,**v**A,**v**B	Simple	Vector Compare Greater-Than Signed Word [Record]
97	vcmpgtfp[.]	**v**D,**v**A,**v**B	Simple	Vector Compare Greater-Than Float [Record]
Vector Compare Equal-To				
98	vcmpequb[.]	**v**D,**v**A,**v**B	Simple	Vector Compare Equal-To Unsigned Byte [Record]
99	vcmpequh[.]	**v**D,**v**A,**v**B	Simple	Vector Compare Equal-To Unsigned Halfword [Record]
100	vcmpequw[.]	**v**D,**v**A,**v**B	Simple	Vector Compare Equal-To Unsigned Word [Record]
101	vcmpeqfp[.]	**v**D,**v**A,**v**B	Simple	Vector Compare Equal-To Float [Record]

Table 13-4. Supported Vector Instructions (Page 5 of 7)

Number	Mnemonic	Operands	Ex-Unit	Description
Vector Compare Greater-Than-or-Equal-To				
102	vcmpgefp[.]	**v**D,**v**A,**v**B	Simple	Vector Compare Greater-Than-or-Equal-To Float [Record]
Vector Compare Bounds Float				
103	vcmpbfp[.]	**v**D,**v**A,**v**B	Simple	Vector Compare Bounds Float [Record]
Vector Conditional Select				
104	vsel	**v**D,**v**A,**v**B,**v**C	Simple	Vector Conditional Select
Vector Pack				
105	vpkuhum	**v**D,**v**A,**v**B	Permute	Vector Pack Unsigned Halfword Unsigned Modulo
106	vpkuhus	**v**D,**v**A,**v**B	Permute	Vector Pack Unsigned Halfword Unsigned Saturate
107	vpkshus	**v**D,**v**A,**v**B	Permute	Vector Pack Signed Halfword Unsigned Saturate
108	vpkshss	**v**D,**v**A,**v**B	Permute	Vector Pack Signed Halfword Signed Saturate
109	vpkuwum	**v**D,**v**A,**v**B	Permute	Vector Pack Unsigned Word Unsigned Modulo
110	vpkuwus	**v**D,**v**A,**v**B	Permute	Vector Pack Unsigned Word Unsigned Saturate
111	vpkswus	**v**D,**v**A,**v**B	Permute	Vector Pack Signed Word Unsigned Saturate
112	vpkswss	**v**D,**v**A,**v**B	Permute	Vector Pack Signed Word Signed Saturate
113	vpkpx	**v**D,**v**A,**v**B	Permute	Vector Pack Pixel32
Vector Unpack High				
114	vupkhsb	**v**D,**v**B	Permute	Vector Unpack High Signed Byte
115	vupkhsh	**v**D,**v**B	Permute	Vector Unpack High Signed Halfword
116	vupkhpx	**v**D,**v**B	Permute	Vector Unpack High Pixel16
Vector Unpack Low				
117	vupklsb	**v**D,**v**B	Permute	Vector Unpack Low Signed Byte
118	vupklsh	**v**D,**v**B	Permute	Vector Unpack Low Signed Halfword
119	vupklpx	**v**D,**v**B	Permute	Vector Unpack Low Pixel16
Vector Merge High				
120	vmrghb	**v**D,**v**A,**v**B	Permute	Vector Merge High Byte
121	vmrghh	**v**D,**v**A,**v**B	Permute	Vector Merge High Halfword
122	vmrghw	**v**D,**v**A,**v**B	Permute	Vector Merge High Word
Vector Merge Low				
123	vmrglb	**v**D,**v**A,**v**B	Permute	Vector Merge Low Byte
124	vmrglh	**v**D,**v**A,**v**B	Permute	Vector Merge Low Halfword
125	vmrglw	**v**D,**v**A,**v**B	Permute	Vector Merge Low Word
Vector Splat				
126	vspltb	**v**D,**v**B,UIM	Permute	Vector Splat Byte
127	vsplth	**v**D,**v**B,UIM	Permute	Vector Splat Halfword
128	vspltw	**v**D,**v**B,UIM	Permute	Vector Splat Word

Table 13-4. Supported Vector Instructions (Page 6 of 7)

Number	Mnemonic	Operands	Ex-Unit	Description
Vector Splat Immediate Signed Integer				
129	vspltisb	**v**D,SIM	Permute	Vector Splat Immediate Signed Byte
130	vspltish	**v**D,SIM	Permute	Vector Splat Immediate Signed Halfword
131	vspltisw	**v**D,SIM	Permute	Vector Splat Immediate Signed Word
Vector Permute				
132	vperm	**v**D,**v**A,**v**B,**v**C	Permute	Vector Permute
Vector Shift Left Double by Octet Immediate				
133	vsldoi	**v**D,**v**A,**v**B,SH	Permute	Vector Shift Left Double by Octet Immediate
Vector Shift by Octet				
134	vslo	**v**D,**v**A,**v**B	Permute	Vector Shift Left by Octet
135	vsro	**v**D,**v**A,**v**B	Permute	Vector Shift Right by Octet
Vector Maximum				
136	vmaxub	**v**D,**v**A,**v**B	Simple	Vector Maximum Unsigned Byte
137	vmaxsb	**v**D,**v**A,**v**B	Simple	Vector Maximum Signed Byte
138	vmaxuh	**v**D,**v**A,**v**B	Simple	Vector Maximum Unsigned Halfword
139	vmaxsh	**v**D,**v**A,**v**B	Simple	Vector Maximum Signed Halfword
140	vmaxuw	**v**D,**v**A,**v**B	Simple	Vector Maximum Unsigned Word
141	vmaxsw	**v**D,**v**A,**v**B	Simple	Vector Maximum Signed Word
142	vmaxfp	**v**D,**v**A,**v**B	Simple	Vector Maximum Float
Vector Minimum				
143	vminub	**v**D,**v**A,**v**B	Simple	Vector Minimum Unsigned Byte
144	vminsb	**v**D,**v**A,**v**B	Simple	Vector Minimum Signed Byte
145	vminuh	**v**D,**v**A,**v**B	Simple	Vector Minimum Unsigned Halfword
146	vminsh	**v**D,**v**A,**v**B	Simple	Vector Minimum Signed Halfword
147	vminuw	**v**D,**v**A,**v**B	Simple	Vector Minimum Unsigned Word
148	vminsw	**v**D,**v**A,**v**B	Simple	Vector Minimum Signed Word
149	vminfp	**v**D,**v**A,**v**B	Simple	Vector Minimum Float
Vector Estimate Float				
150	vrefp	**v**D,**v**B	Float	Vector Reciprocal Estimate Float
151	vrsqrtefp	**v**D,**v**B	Float	Vector Reciprocal Square Root Estimate Float
152	vlogefp	**v**D,**v**B	Float	Vector Log 2 Estimate Float
153	vexptefp	**v**D,**v**B	Float	Vector 2 Raised to the Exponent Estimate Float
Vector Negative Multiply-Subtract Float				
154	vnmsubfp	**v**D,**v**A,**v**C,**v**B	Float	Vector Negative Multiply-Subtract Float
Vector Round to Floating-Point Integral Value				
155	vrfin	**v**D,**v**B	Float	Vector Round to Floating-Point Integer Nearest
156	vrfiz	**v**D,**v**B	Float	Vector Round to Floating-Point Integer toward Zero

Table 13-4. Supported Vector Instructions (Page 7 of 7)

Number	Mnemonic	Operands	Ex-Unit	Description
157	vrfip	**v**D,**v**B	Float	Vector Round to Floating-Point Integer toward Positive infinity
158	vrfim	**v**D,**v**B	Float	Vector Round to Floating-Point Integer toward Minus infinity
Vector Convert To Fixed-Point				
159	vctuxs	**v**D,**v**B,UIM	Float	Vector Convert to Unsigned Fixed-Point Word Saturate
160	vctsxs	**v**D,**v**B,UIM	Float	Vector Convert to Signed Fixed-Point Word Saturate
Vector Convert From Fixed-point				
161	vcfux	**v**D,**v**B,UIM	Float	Vector Convert From Unsigned Fixed-Point Word
162	vcfsx	**v**D,**v**B,UIM	Float	Vector Convert From Signed Fixed-Point Word